THE SEMANTICS OF PASCAL IN LCF

Luigia Aiello, et al

Stanford University

Prepared for:

Office of Naval Research
Advanced Research Projects Agency

August 1974

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD787 631

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: STAN-CS-74-447	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): THE SEMANTICS OF PASCAL IN LCF	5. TYPE OF REPORT & PERIOD COVERED: technical, August 1974
6. PERFORMING ORG. REPORT NUMBER: STAN-CS-74-447	
7. AUTHOR(s): L. Aiello, M. Aiello, and R. W. Weyhrauch	8. CONTRACT OR GRANT NUMBER(s): DAHC 15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University, Computer Science Department, Stanford, California 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: ARPA/IPT, Attn: S. D. Crocker, 1400 Wilson Blvd., Arlington, Va. 22209	12. REPORT DATE: August, 1974
	13. NUMBER OF PAGES: ~~78~~ 80
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): ONR Representative: Philip Surra, Durand Aeronautics Bldg., Rm. 165, Stanford University, Stanford, California 94305	15. SECURITY CLASS. (of this report.): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitations on dissemination.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We define a semantics for the arithmetic part of PASCAL by giving it an interpretation in LCF, a language based on the typed λ-calculus. Programs are represented in terms of their abstract syntax. We show sample proofs, using LCF, of some general properties of PASCAL and the correctness of some particular programs. A program implementing the McCarthy Airline reservation system is proved correct.

80

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM No.221

COMPUTER SCIENCE DEPARTMENT
REPORT No.447

AUGUST 1974

The Semantics of PASCAL in LCF

by
Luigia Aiello
Mario Aiello
and
Richard W. Weyhrauch

Abstract:

We define a semantics for the arithmetic part of PASCAL by giving it an interpretation in LCF, a language based on the typed λ-calculus. Programs are represented in terms of their abstract syntax. We show sample proofs, using LCF, of some general properties of PASCAL and the correctness of some particular programs. A program implementing the McCarthy Airline reservation system is proved correct.

Authors' addresses

L. Aiello, Istituto di Elaborazione dell'Informazione, via S. Maria 46, 56100 Pisa, Italy;

M. Aiello, Istituto di Scienze dell'Informazione, Universita' di Pisa, corso Italia 40, 56100 Pisa, Italy;

R. Weyhrauch, AI Lab., Computer Science Dept., Stanford University, Stanford, California 94305. or Weyhrauch @SU-AI

This research is supported (in part) by the Advanced Research Projects Agency of the Office of the Secretary of Defense (DAHC 15-73-C-0435).

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, or the U.S. Government.

Reproduced in USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

The Semantics of PASCAL in LCF

TABLE OF CONTENTS

SECTION 1 INTRODUCTION

This paper is an attempt to determine the order of magnitude of the problem of giving an axiomatic treatment, in LCF, of an established programming language with a sizable user community. We wanted to include such features as declarations, I/O, different types of parameter bindings and control structures. For this purpose we chose the integer arithmetic part of PASCAL, which we will refer to as PASCAL. It seemed to us a reasonable choice in that:

1) it satisfies the above criterion, thus it is not a *toy* language.

2) it is powerful enough to compute any partial recursive function on sequences of integers.

3) the existence of VCGEN (Igarashi, London and Luckham 1973) and FOL (Weyhrauch and Thomas 1974) will eventually give us the ability to compare the effectiveness of Hoare's axiomatic definition of PASCAL, McCarthy's style of first order axiomatization (McCarthy and Painter 1966) and the Scott style of assigning extensional meanings to programs.

One pleasant result of our work was the discovery that the task seems more manageable than we had originally thought. Most discouraging was realizing exactly how inadequate even careful descriptions of programming languages actually are.

LCF is both a logical calculus and a proof-checker for a suspected proof in the logic. It could be described as an equation calculus based on terms in the typed λ-calculus, whose most powerful rule of inference is Kleene's first recursion theorem stated as a rule (see Kleene 1952). Using this language in the mathematical theory of computation was first suggested by Dana Scott. Its formal properties are described in Milner 1972a, 1972b. Also see Milner and Weyhrauch 1972, Weyhrauch and Milner 1972, Newey 1973, 1974, Aiello and Aiello 1974 for other applications. A short description of LCF syntax is given in appendix 1.

Initially our intent was to present a semantics for the description of PASCAL given in Wirth 1971, 1972 and Wirth and Hoare 1973. As a result of our attempts to give what *we* consider a complete description, we found many ambiguities and places where the literal interpretation of Wirth's descriptions led to a semantics having undesirable properties (see 3.3.2.3 for a discussion of the for statement). We have described a language which has a fairly smooth semantics, and whose formal properties are more clearly apparent. All the differences are documented in the text.

We think of our axiomatization as characterizing properties of the *whole* PASCAL and not as a description of properties of individual statements. In section 4.2, for instance, we prove that, if two programs P and Q don't contain goto statements, we can represent the function computed by the program consisting of P appended to Q as the composition of the function computed by P with that computed by Q. This theorem and others in section 4 simply cannot be expressed or used in formalisms like Floyd's method of attaching assertions to programs or in Hoare's axiomatic approach. We consider this a major difficulty with those techniques. Both consider programs individually. It is our belief that the feasibility of checking (or generating) large formal proofs depends on our ability to prove general properties of classes of programs. A description of the entire programming language is required in order to mention these classes.

Characterizing an entire language in this way means that conflicts arising out of putting different

programming features together must be resolved, or at least describable in the formalism. The discussion of function activations in section 3.2.1.3 is a typical example of the difficulty one encounters when trying to characterize the behavior of an entire language. Unusual programs cannot be ignored or left unmentioned. In actual programming languages the ability to decide if a program is well formed is in general too costly and many "ill formed" programs are usually accepted by the parser. An example of such a difficult case is found in section 3.3.2.3. on the for statement.

In section 2 we describe the axiomatization of the environment in which PASCAL programs are *executed*.

A special word is needed here to make clear an abuse of language that appears throughout the report. We frequently speak about a combinator being executed and then explain what it does. Strictly speaking this is not correct. Combinators don't *do* anything. The functions we mention are to be interpreted extensionally. It means that the only properties of LCF functions that can be mentioned are properties of their graphs. Thus, when looking at

F = [λN.(isname(N)→(isRichard(N)→Good,Bad),FF)]

we may say informally that F is a function which checks if N is a name. If it is not then its value is FF otherwise it returns Good or Bad depending on whether that name is Richard or not. This description is in the style of an interpreter. More correctly we should say, F is a three valued function whose value is FF on arguments which are not names, and otherwise has the value Good or Bad depending on whether that name is Richard. *How* the function is *computed* is transparent to LCF. This point is very important so that there is no confusion about the nature of the semantics defined here. To each program is assigned a function, *not* a computation procedure. LCF terms also have interpretations as computation procedures, but it is not this interpretation that concerns us here.

Section 3 describes all the control structures and statements relevant to the arithmetic part of PASCAL. They include

1) type definitions
2) variable and array declarations,
3) procedure declarations and procedure activations,
4) function declarations and function evaluations,
5) assignment, conditional, while, repeat, for-to,for-downto and goto statements,
6) input/output instructions.

We do not consider constant definitions, label declarations (Wirth 1972), case or with statements, or records and files (except INP and OUT). These are either easily addable or are not relevant to the arithmetic part of PASCAL.

Although LCF uses the typed λ-calculus, a natural semantics may be given to goto's and to procedures having themselves as actual parameters without introducing type conflicts. This is explained in section 3.3.1.3.

Examples of general theorems about PASCAL are presented in section 4. Most of the work to date on the correctness and equivalence of *programs*, has actually only dealt with the extensional

properties of algorithms. Input/output or the effects of declarations cannot be ignored in any theory of correctness which hopes to be practical. As soon as we ask whether a program will run or not, or whether it will compile or not, then the question "do we have the correct algorithm?" is a minimal criterion for correctness. In addition, the distribution and consumption of resources during the execution of a program, involves both what has been declared and how bindings are made to parameters. The correctness of programs which input data incrementally, must know how these inputs are treated.

We have set out here a description of a large but stable core for any interesting programming language. We wanted to establish a base from which further work could be done towards a practical system for proving properties of programs within this core. Some example are the theorems of section 4.

Section 5 gives partial correctness proofs for some programs. The much overworked factorial program is again discussed. We included it to show some of the flexibility in our approach to program correctness as well as illustrate points made in other parts of the report. A proof of the correctness of a program implementing the McCarthy Airline reservation system is given. This is new in that it treats an interactive program which has a potentially infinite number of inputs. The details are in 5.2.

The appendices contain a short description of the LCF syntax, the list of all the LCF axioms describing the syntax and semantics of PASCAL, and the actual LCF printouts of the proofs of theorems mentioned in the text.

Some familiarity with the papers Wirth 1971, 1972 and Wirth and Hoare 1973 is recommended to better understand this memo.

SECTION 2 THE SEMANTICS OF PASCAL

Section 2.1 Description of the semantics

In this version of PASCAL we restrict our attention to programs whose inputs are sequences of integers. The meaning (or interpretation) we assign to a program is thus a function from sequences of integers into sequences of integers.

Programs, on the other hand, map memories onto memories. In order to describe the effects of procedures and function activations more clearly we introduce the notion of a *store*. A *store* divides the memory into *frames* or environments. *Frames* are specified by a *framepointer*. Thus we think of programs as mapping *stores* onto *stores*, and *stores* are functions from *framepointers* to *frames*.

store: framepointer → frame

A *frame* is a function from *locations* to *values*.

frame: location → value

A *store* describes abstractly additional structure of a memory without knowing how it is realized in any particular implementation. The execution of a program, p, starts with the creation of the initial store. This is done by FRAME0 (see next section). It contains the locations fileloc INP and fileloc OUT for the input and output files respectively, and a location textloc where the text of the program is stored. This *store* has only one *frame* called 0.

Type definitions are then made in this *frame*. Each *frame* represents an environment in which the current declarations and variable bindings are found.

The effect of declaring a variable, v, in a *frame* is to create a location typeloc v, which contains the type of v. Thus we can tell if a variable has been declared in a frame s(f) by checking if

s(f,typeloc v)=UNDEF.

The execution of a procedure or a function creates a new *frame*. It is set up by the combinator MAKFRAME defined in appendix 3.9. The new *framepointer* is just the successor of the current one, namely that pointing to the frame where the procedure or function has been activated. This imposes a stack discipline on procedure and function activations. The binding of free variables are made in the style of ALGOL. The position of the variable declaration in the program text determines the binding frame. FETCHV is the function which looks up the value currently bound to a variable.

The combinators FRAME0 and MAKFRAME build *stores* with the following property. If f is a *framepointer* corresponding to a non activated frame, then s(f)=UU, otherwise for any legal location loc, s(f,loc) is either a value or is UNDEF. The value of a variable is stored in a location which depends on its name. This is slightly complicated in PASCAL, because both identifiers and array element names (e.g. A[1]) are considered variables. Section 3.2.1.2 describes the combinators which allow us to treat them uniformly.

Both FRAME0 and MAKFRAME store the body of statements to be evaluated into a location of the frame they are defining. The effect of procedure and function declarations is to add new locations to the store.

The statement part of a program, procedure or function, is interpreted in the store where the corresponding declaration part has been evaluated. Statements are evaluated in sequential order, unless a goto statement is encountered. Where to go is determined by the function **segm**, which takes a text and a label, and returns a text, i.e. it tells you where to jump. The new text is evaluated in the same *frame* as you jumped from. Thus you cannot jump out of a procedure activation. This follows Wirth 1971. The effects of the other statements are pretty much as you might expect. They are defined by MS in section 3.3.

The stack discipline imposed on procedure and function activations and the discipline imposed on goto's are not intrinsic to this approach to the description of the semantics of programming languages. We impose them because we wanted to correspond to Wirth 1971.

Programs are written in abstract syntactic form. Each syntactic construct is assembled by a *constructor* and its components are selected by a *selector*. The list of all the axioms about the syntactic constructors and selectors are given in appendices 2.1 and 2.2. Each construct is identified by associating a type to it. A predicate is defined which is satisfied only by objects of that type (see appendix 2.3). The equality of identifiers denoting types of syntactic constructs and of location names is denoted by "=" in the formulas through the text and is detected by LCF itself.

Section 2.2 Top level functions

The function FUNCT:

```
FUNCT ≡ [λp o.[λi.(INPUT⊗PASCAL(p,o)⊗OUTPUT)(i)]] .
```

where ⊗≡[λf g x.g(f(x))] is the composition function and i, o are sequences of integers, represents the "interface" between functions which compute on integers and programs which compute on stores.

Wirth 1971 describes a program as a PASCAL procedure which has an input and an output file as parameters. The combinator PASCAL

```
PASCAL ≡ [λp.[λo i.MP(p,0,FRAME0(p,o,i))]]
```

when applied to a program, p, is a function which takes as arguments two sequences of integers o and i (representing the initialization of the output and input files respectively) and returns a function from stores to stores. The definition of PASCAL imitates explicitly the bindings which a procedure would make when executed as part of a program. FRAME0(p) applied to o and i creates a store containing a single frame, called 0, with these bindings and then applies MP to the program p in frame 0 and this store.

```
FRAME0 ≡ [λp.[λo i.[λf. (f=0) →
                    [λloc.(loc=fileloc !NP)→ INTERNALREP(i),
                         (loc=fileloc OUT)→ INTERNALREP(o),
                         (loc=textloc)→ statmof(p),UNDEF],UU]]],
```

PASCAL programs read sequences of numerals supplied by some input device into the buffer filoloc INP and write outputs into the buffer filoloc OUT. INPUT is just the identity function. The write statement puts numerals in the output buffer, thus OUTPUT maps sequences of numerals, onto sequences of integers. INTERNALREP is a function which takes sequences of integers and returns sequences of numerals. The definitions are found in appendix 3.1.

Programs in PASCAL have two parts: a declaration part and a statement part.

The interpretation of a program in some frame specified by the framepointer f:

MP = [λp f.MD(declof f,f)⊛MS(statmof f,f)]

is just the interpretation of definitions MD composed with that of statements MS. These are described in the next section.

SECTION 3 DESCRIPTION OF THE LANGUAGE

This section contains the description of all the instructions included in our version of PASCAL and the description of their semantics in LCF. Each text (it may be a program, a procedure or a function text) consists of two parts: declaration part and statement part. The semantics of a text depends on the frame in which such text is executed, for this reason a framepointer is specified as parameter in every semantic function.

Section 3.1 Declaration part

The declaration part includes *type definitions* and the declaration of all the variables, functions and procedures local to that text. Its semantics is defined by:

```
MD   ≡ [λd f.MDEF(d,f)⊗MDEC(d,f)],

MDEF ≡ [∝F.[λd f.
            isemptyst  d → ID,
            istypedef  d → CREAT(f,namof d,typof d),
            iscmpnd    d → F(fstof d,f)⊗F(rmdof d,f),ID]],

MDEC ≡ [∝F.[λd f.
            isemptyst  d → ID,
            isvardecl  d → CREAV(f,namof d,typof d,f),
            isprocdecl d → CREAP(f,namof d,prspof d,f),
            isfundecl  d → CREAF(f,namof d,fnspof d,typeof d,f,f),
            iscmpnd    d → F(fstof d,f)⊗F(rmdof d,f),ID]].
```

MD is the composition of MDEF, which defines the semantics of type definitions and MDEC, which defines the semantics of variable, procedure and function declarations. Every identifier appearing in a declaration statement is a name so it must satisfy the predicate isname. Consequently, whenever some property of a PASCAL program is to be proved in LCF, for each identifier appearing in that program, axioms stating that it is a name are to be added. The predicates for the identification of syntactic constructs are given in appendix 2.3.

3.1.1 Data Type Definitions

Since we are dealing with the integer arithmetic part of PASCAL, the *scalar data types* we have introduced are the integer type INT and its subranges. A subrange is an interval of integers and is defined by specifying its lower and upper bounds. The *structured data types* included in our language are the array types. An array may have any number of indices (each ranging in a subrange type) and its elements are all of the same scalar type.

Each type may be assigned a name in a *type definition*. The semantics of a type definition is CREAT:

```
CREAT ≡ [λf n ty s.CREALOC(f,s,typidloc,n,ty)].

CREALOC ≡ [λf s loc n val. ISPRESENT(n,s(f))→UU,STORE(f,s,loc n,val)]
```

CREALOC is used by CREAT. It declares a name n to be a synonym for the type ty in the frame s(f), by storing ty in a new location typidloc n. The result of CREALOC is undefined if n doesn't satisfy the predicate isname or if it has been already declared in the current frame. This is tested by ISPRESENT. Modification of the store is done by the combinator STORE. Their definitions are in appendix 3.9.

In the definitions of MDEF and CREAT no assumption is made on the order of the type definitions. If all the type identifiers satisfy the predicate isname and are different from each other, the result of MDEF on a frame, in which they don't appear, doesn't depend on their order in the text (see theorems in 4.5)

3.1.2 Variable Declarations

Each variable occurring in a text must be assigned a type which specifies the range of values that variable may assume during the execution of the statement part of the text. The semantics of a variable declaration is defined by CREAV:

CREAV = [λf n ty fl s.CREALOC(f,s,typeloc,n,TYPEVAL(ty,fl,s))].

CREAV creates a location in the current frame s(f), whose name is typeloc n, provided n is a name and no other location with the same name already exists in that frame. The content of that location is the type associated with n. Such type is evaluated by TYPEVAL (see 3.3.1.3). Each type identifier possibly appearing in it is removed and its definition is substituted for it. The evaluation is made in the frame specified by the framepointer fl. When a variable is declared fl coincides with f, so at the moment there is no point in introducing another parameter in CREAV. We have introduced this extra parameter since CREAV is also used when binding value parameters in a procedure or function activation. On that occasion the two framepointers f and fl (the one in which the new location is created and the one in which the type evaluation starts) do not coincide.

3.1.3 Procedure and Function Declarations

The semantics of a procedure declaration is defined by CREAP:

CREAP = [λf n ps fl s.STORE(f,CREALOC(f,s,acclnk,n,fl),procloc n,ps)],

The result of CREAP is undefined if n is not a name or something with the same name has already been declared. Otherwise two locations are created. One of them, whose name is procloc n contains the formal argument list and the text associated to that procedure declaration, the other one, whose name is acclnk n contains the frame pointer specifying the frame where the procedure has been declared, i.e. the environment where its free variables are bound. As for variable declarations, when a procedure is declared the two framepointers f and fl are the same, but the combinator CREAP is also used when binding procedure parameters in a procedure or function activation, and in that case the two framepointers differ.

The semantics of a function declaration is CREAF:

```
CREAF = [λf n fs ty ft fl s.
        STORE(f,STORE(f,CREALOC(f,s,acclnk,n,fl),typeloc n,TYPEVAL(ty,ft,s)),funcloc n,fs)].
```

CREAF is similar to CREAP. The only difference is that, in addition to funloc n and acclnk n, a location typeloc n is created, whose content is the type of the result of that function.

From the definition of MDEC and the others LCF combinators describing the semantics of the declarations it follows that the order in which declarations are made is not relevant. If the identifiers being declared are different and no other locations have been declared with these names the same store is obtained, independently of the order (see theorems in 4.5). This is slightly more general than the definition of PASCAL in Wirth 1971, which requires that all the variable declarations must appear before the function and procedure declarations.

Section 3.2 Expressions

An LCF function can either evaluate to an object or to a truth value, but not both. For this reason we could not introduce a unique evaluation function for arithmetic and boolean expressions. So we have divided expressions into arithmetic and boolean (this distinction is absent in Wirth 1971) and introduced two evaluation functions. Furthermore, we have introduced a finer distinction between the types of operators in order to avoid funny situations like the prefix adding operator "*or*" which is allowed in the syntax given in Wirth 1971, 1972 but whose meaning is not defined there.

3.2.1 Arithmetic Expressions

Arithmetic expressions are written in abstract syntactic form and are evaluated by MEXPR:

```
MEXPR = [∝F.[λe f s.
     isconst e    → MCONST e,
     isexpr e →isunary(opof e) → MOP1(opof e,F(arg1of e,f,s)),
               isbinary(opof e)→ MOP2(opof e,F(arg1of e,f,s),F(arg2of e,f,s)),
     isvariable e → FETCHV(e,f,s),
     isfundes e   → RETURN(succ f,MF(namof e,actargof e,f,s)),UU,UU]].
```

3.2.1.1 Evaluation of Constants and Expressions

The abstract syntactic representation of numbers is defined by the combinator mknumconst. If n is a number, mknumconst n is the corresponding numeral and it satisfies the predicate isconst (see appendix 2.3). Numerals are evaluated by the semantic combinator MCONST, which returns the corresponding number.

```
MCONST = [λx.isconst x → numof x,UU].
```

Arithmetic operator symbols appear explicitly in expressions and satisfy the predicate isunary or isbinary according to the number of arguments the corresponding operator expects (see definitions in appendix 2.4). When evaluating arithmetic expressions MEXPR checks whether the operator symbol is unary or binary, then MOP1 or MOP2 evaluates them and applies the corresponding value to the argument(s) evaluated recursively.

```
MOP1 = [λx.x=pplus→λx.x,x=pminus→λx.(0-x),x=plus1→succ,x=minus1→pred,UU].
```

```
MOP2 = [λx.x=plus→!+,x=minus→!-,x=times→!*,x=div→!/,x=rmdr→mod,UU].
```

MOP1 evaluates unary operator symbols and MOP2 evaluates binary operator symbols to the corresponding functions. For example, the meaning of the symbol plus is the LCF function +. Note that, due to the LCF syntax, infix operators, when written without arguments, are prefixed by "!". An LCF axiomatization of arithmetic is given in Newey 1973.

As an example, if:

```
mkexpr2(plus,mkexpr1(plus1,n1),mkexpr2(times,mknumconst 2,mkexpr1(minus1,n2)))
```

is evaluated in a frame where the location n1 contains the value 3 and the location n2 contains the value 7, its result is 16, i.e. succ(3)+(2*pred(7)).

3.2.1.2 Evaluation of Variables

If the expression to be evaluated is a variable, then the corresponding value is fetched by the FETCHV combinator.

```
FETCHV = [∝F.[λn f s.
    ISLOCAL(typeloc NAMOFVAR(n),s(f))→ISLOCAL(NAMOFVAR(n),s(f))→s(f,LOCOFVAR(n,f,s)),UU,
    istopf(f)→UU,F(VARBNDTO(n,f,s),NEWFP(n,f,s),s)]].
```

The fetching mechanism is very simple. The variable to be fetched may be an entire variable of a scalar type or an array element. In both cases a test is done (by ISLOCAL) to see whether or not that variable name has been declared in the current frame. If this is the case, the corresponding value is fetched in the current frame (it will be undefined if the variable has been declared, but no value has been assigned to that location). If the variable name has not been declared in the current frame and the current frame is not the top one (i.e. if the fetching is done during a procedure or function activation), the binding list is checked. In fact the variable to be fetched may be a formal parameter passed by name (see 3.3.1.3 for details on the binding mechanism). In this case FETCHV applies recursively to the corresponding actual parameter in the preceding frame. If that variable name is not found in the binding list, the variable is free for that procedure or function activation, hence FETCHV applies recursively to the same variable in the frame specified by the result of NEWFP, i.e. the frame where the procedure or function in execution has been declared, hence where its free variables are bound.

The definitions of the auxiliary combinators used in FETCHV may be found in appendix 3.7,-9. ISLOCAL performs a test to see whether a given name has been declared or not in a frame. NAMOFVAR applies to a variable n, and gives as result its name: it coincides with n if n is an entire variable of scalar type, or it is the name part of n if n is an array element. Analogously LOCOFVAR returns the location of n. As above, the location of n might be n itself, or an array location. varbndfo is the function which accesses the list of parameter bindings. If the variable n appears in it, then n (or its name-part) is a formal name parameter and the corresponding actual parameter is the result of varbndto. If n is not a name parameter, then n itself is the result of varbndto. In this case n is a free variable for the function or procedure in execution. NEWFP evaluates to pred f or to the content of the alnk location of the current frame, according to whether n is a formal parameter or a free variable. The alnk location is set up when a new frame is created for a procedure (function) activation, it contains the pointer to the frame where the activated procedure (function) has been declared.

From the definition of NAMOFVAR given in appendix 3.7 we see that its result is undefined if it is applied to FUNV. As explained in 3.2.1.3 and 3.3.1.2 FUNV is the location where the value of a function is stored. Since NAMOFVAR is undefined on FUNV, the result of FETCHV is undefined if it applies to FUNV. So it is impossible to "read" the value of a function with the usual fetching combinator.

3.2.1.3 Function Designators

If the expression to be evaluated is a function designator, then a new frame is set up. The function is evaluated by MF and its value is retrieved by the RETURN combinator in a special location named FUNV.

RETURN ≡ [λf s.ISLOCAL(FUNV,s(f))→s(f,FUNV),UU],

The semantics of a function activation is very similar to that of a procedure activation (see 3.3.1.3). Starting from a given store, a new frame is created by the combinator MFB and then the semantic function MP (described in section 2.2) is applied to the text of the function. The current frame is changed by incrementing the frame pointer by 1.

MF ≡ [λn a f. MFB(FUNCFAL(n,f),a,f,n)⊙MP(FUNCDEF(n,f),succ f)].

FUNCFAL and FUNCDEF are the two functions which fetch from the store the formal argument list and the text of the function being activated. Their definition is given in appendix 3.8. They use the FETCH combinator which, like FETCHV, returns the content of a location from the frame where it has been created.

The activation of a new frame and the binding of parameters is done by MFB:

MFB ≡ [λfa aa f n s.BIND(fa,aa,succ f,CREALOC(succ f,typeloc FUNV,TYPEDEF(n,f,s),
MAKFRAME(FUNCBODY(n,f,s),PFLNK(n,f,s),succ f,s)))].

It not only binds the formal parameters to the actual parameters (the binding function BIND will be fully explained in 3.3.1.3), but it also creates a new frame. The frame in which the function is evaluated is set up by MAKFRAME (see appendix 3.9). It creates a location textloc where the statement part of the text is stored, and a location alnk whose content is a pointer to the frame where the function has been declared. Moreover, a location typeloc FUNV is created, whose content is the type of the function being evaluated. A location named FUNV will eventually contain the value of the function. In fact Wirth 1971, 1972 says that the function name must appear at least once in the function text at the left hand side of an assignment statement. The value of the function in execution is stored in the FUNV location by the combinator ASSIGN. From its definition in 3.3.1.2 we see that the result of a function can only be assigned to FUNV in its function frame. This means that if the name of the function in execution appears at the left hand side of an assignment statement in the text of a procedure where such identifier has not been declared, it is interpreted as a free variable, not the name of the function in execution.

As noted in 3.2.1.2 the FETCHV combinator returns an undefined value if applied to FUNV. This implies that a variable named FUNV cannot be declared even in a frame different from that set up by a function activation. We have prevented this by considering FUNV a "reserved" identifier which

doesn't satisfy the predicate **isname**, so it cannot be used in declarations (the axiom **isname FUNV=FF** is included in appendix 2.4).

We assume that the translator from concrete to abstract syntax has substituted **FUNV** for all the occurrences of the function name on the left hand side of assignment statements within the function text. If there are no such occurrences, the function activation returns an undefined result. If there are several, the last executed determines the value of the function. If a variable identifier equal to the name of the function in execution occurs on the rigth hand side of an assignment statement, then either that variable has been declared within the function execution or it is considered a free variable of that function. When a variable has been declared with the same name as the function in execution, its value is undefined during the function execution. In fact, it cannot be assigned a value since **FUNV** has replaced it on the left hand side of any assignment statement. It cannot be inputed since the read statement cannot be executed within a function activation (see the following paragraphs for a discussion on side effects).

The declaration of a variable with the same name as the function in execution is not forbidden by Wirth 1971, 1972, but we do not see any reasonable semantics for it. In addition Wirth 1971, 1972 says that:

> *"Occurrence of the function identifier in a function designator within its declaration implies recursive execution of the function".*

This sentence doesn't specify what happens if within a function another function is declared with the same name. Our semantics allows such declarations - why not? In such case the "outermost" function cannot be executed recursively. This is also the case if a function has a formal parameter with the same name (this is not forbidden in Wirth 1971, 1972). In this case the corresponding actual parameter is executed.

PASCAL allows functions to have themselves as actual parameters. Even though LCF is a typed logic, the semantic combinators we have defined avoid type conflicts by passing the text of the function and not the function itself as a parameter. This is also true for procedures having themselves as parameters.

Haberman 1973 is very critical of the PASCAL's notion of function. He says that, while the aim of a PASCAL function is that of not having side effects, this is not true since a function may call a procedure which may have side effects. Our semantics deals with this situation in a different way. Statements which change the content of a location and hence cause side effects are only the assignment, read, write and for statements.

The read and write statements modify the content of the input and output buffers so they cannot be executed during a function activation. We forbid this by the test **ISFUNFP** which is performed whenever a read/write statement is executed. It checks if any frame between the current one and the top one has been set up by a function activation (see 3.3.1.4,-5). The test on whether a frame has been created for a function activation or for a procedure activation is done by checking in the frame whether **typeloc FUNV** is defined or not.

An assignment statement may cause side effects by assigning a value to a free variable. Whenever the variable to be assigned is a free variable for the current frame, the **ASSIGN** combinator (see

3.3.1.2), checks whether between the current frame and that where the variable is bound (hence where the modification of the store actually takes place) a function has been activated.

The for statement may cause a side effect if its control variable is free in a function activation. Wirth 1971, 1972 doesn't say that the control variable must be local to the frame where the for statement is executed. In our semantic definition of PASCAL, the for statement cannot cause side effects in a function activation since its definition relies on the combinator ASSIGN for updating the control variable (see 3.3.2.3).

We included the above checks in our semantics so that ill-formed programs return an undefined store. It turns out, however, that in our formalism no function can cause side effects. This is because MEXPR simply returns a value from a function activation. The checks done in our semantic combinators amount to checking for side effects "at run time". Thus some programs which would be rejected by a PASCAL compiler will still have well defined meaning for us if the statements producing side effects are never executed.

Finally, we want to point out that our semantics allows parameters of a function to be passed by name, but guarantees that those parameters can only be "read" during the function execution. This contrasts with Hoare's opinion (private communication) that PASCAL functions must not have parameters passed by name. Wirth 1971, 1972 says nothing about it. In Wirth 1971 the assignment to nonlocal variables is explicitly forbidden. Nothing is said about this in Wirth 1972.

3.2.2 Boolean Expressions

The evaluation of boolean expressions is very similar to that of arithmetic expressions (see 3.2.1 and subsections). It is performed by MBEXPR:

```
MBEXPR ≡ [∝F.[λe f s.
   (e=true)→TT,
   (e=false)→FF,
   isbexpr e →isbunary(bopof e) → MBOP1(bopof e,F(barg1of e,f,s)),
              isbbinary(bopof e)→ MBOP2(bopof e,F(barg1of e,f,s),F(barg2of e,f,s)),
              isrelop(bopof e)  → RELOP(bopof e,MEXPR(arg1of e,f,s),MEXPR(arg2of e,f,s)),UU,UU]].
```

true and false are the abstract syntactic representations of the boolean constants true and false. If the expression to be evaluated is the constant true, then it evaluates to TT, if it is the constant false, it evaluates to FF. Boolean expressions containing unary and binary operator symbols are evaluated like arithmetic ones. Relation operators take integers as arguments, so the meaning of a relation symbol is applied to its arguments evaluated by MEXPR. The meaning of unary and binary boolean operators and that of relation operators is defined by MBOP1, MBOP2 and RELOP:

```
MBOP1  ≡ [λx.x=not→¬,UU],
MBOP2  ≡ [λx.x=and→!∧,x=or→!∨,UU],
RELOP  ≡ [λx.x=leseq→!≤,x=greq→!≥,x=lt→!<,x=gt→!>,x=eq→!=,x=neq→≠,UU].
```

For example in the frame specified by the frame pointer f and in the store s

```
mkbexpr1(not,mkbexpr2(or,mkrel,lt,a,mknumconst 8),mkrel(gt,a,mknumconst 1)))
```

evaluates to

```
¬((MEXPR(a,f,s)<8)∨(MEXPR(a,f,s)>1)).
```

An LCF axiomatization for the boolean operators is given in Newey 1973.

Section 3.3 Statement Part

The semantics of the statement part of the program is defined by MS.

```
MS≡[∝F.[λst f.
  isempfyst st    → ID,
  iscmpnd sf      →
   isemptyst(fstof st)→ F(rmdof st,f),
   islabstaf(fstof st)→ F(mkcmpnd(statmof(fstof st),rmdof sf),f),
   isgoto(fstof sf)   → GOTO(F,labelof(fstof st),f),
   isass (fstof st)   → ASSIGN(lhsof(fstof st),MEXPR(rhsof(fstof sf),f),f)⊗F(rmdof st,f),
   isproccall(fstof st)→[λs.MPB(PROCFAL(namof(fstof st),f,s),actargof(fstof sf),f,s,namof(fstof st))]⊗
                        [λs.MD(PROCDECL(namof(fstof st),f,s),succ f,s)]⊗
                          [λs.F(PROCBODY(namof(fstof st),f,s),succ f,s)]⊗CLEAR(succ f)⊗F(rmdof st,f),
   isread(fstof st)   → READ(namof(fstof sf),f)⊗F(rmdof st,f),
   iswrite(fstof st)  → WRITE(namof(fstof st),f)⊗F(rmdof st,f),
   iscond(fstof st)   → COND(MBEXPR(tostof(fstof st),f),
                        F(append(thenof(fstof st),rmdof sf),f),F(append(elseof(fstof st),rmdof st),f)),
   iswhile(fstof st)  → COND(MBEXPR(testof(fstof st),f),
                        F(append(bodyof(fstof st),st),f),F(rmdof st,f)),
   isrepeat(fstof st) → F(append(bodyof(fstof st),mkcmpnd(mkcond(mkbexpr1(not,
                             testof(fstof sf)),fstof st,ES),rmdof sf)),f),
   isforto(fstof sf)  → COND(MBEXPR(fortest(fstof sf),f),
                        ASSIGN(indexof(fstof sf),MEXPR(lbof(fstof st),f),f)⊗
                        F(append(bodyof(fstof sf),fortoup sf),f),F(rmdof st,f)),
   isfordn(fstof sf)  → COND(MBEXPR(fortest(fstof sf),f),
                        ASSIGN(indexof(fstof sf),MEXPR(ubof(fstof sf),f),f)⊗
                        F(append(bodyof(fstof sf),fordnup sf),f),F(rmdof sf,f)),UU,UU]].
```

The definition of MS has the form of a nested conditional, each branch corresponds to one instruction of the language. Note that MS is defined only on the empty statement ES, whose semantics is the identity ID≡[λx.x], and on compound statements. In fact, the abstract syntactic form of a program is a list of instructions assembled by the constructor mkcmpnd and ending with the empty statement ES. When the first argument of MS is a compound statement a test is done on its first element. Except for the labeled statements, whose semantics is simply that the corresponding unlabeled statement, the detailed description of the semantic functions defining the meaning of each instruction will be given in the following sections.

3.3.1 Simple Statements

We have defined the semantics of all the simple statements of PASCAL, i.e. goto statement, assignment statement, and procedure statement. Furthermore, we have defined the semantics of an instruction for reading input data from the input buffer INP and of an instruction which writes output data into the output buffer OUT.

3.3.1.1 Goto Statement

The semantics of the goto statement is defined by the GOTO combinator.

```
GOTO  = [λF.[λn f. F(segm(n,TEXT(f)),f)]],
```

It applies the semantic function MS recursively to the text returned by the **segm** combinator:

```
segm = [∝F.[λn st.
     isemptyst st → UU,
     iscmpnd st→
      isemptyst(fstof st)  →F(n,rmdof st),
      islabstat(fstof st)→(n=labelof st)→ st,F(n,mkcmpnd(statmof(fstof st),rmdof st)),
      issingle(fstof st) →F(n,rmdof st),
      iscond(fstof st)   →occurs(n,thenof(fstof st))→append(F(n,thenof(fstof st)),rmdof st),
                         occurs(n,elseof(fstof st))→append(F(n,elseof(fstof st)),rmdof st),
                         F(n,rmdof st),
      isrepwh(fstof st)  →occurs(n,bodyof(fstof st))→append(F(n,bodyof(fstof st)),st),
                         F(n,rmdof st),
      isforto(fstof st)  →occurs(n,bodyof(fstof st))→
                          append(F(n,bodyof(fstof st)),fortoup(st)),F(n,rmdof st),
      isfordn(fstof st)  →occurs(n,bodyof(fstof st))→
                          append(F(n,bodyof(fstof st)),fordnup(st)),F(n,rmdof st),UU,UU]].
```

segm applies to a label, and the text st which is retrieved from the store by the TEXT combinator, and returns the piece of text starting from the first occurrence of the label. If the label is not found in the text the result of **segm** is undefined. The behaviour of PASCAL programs when several identical labels appear in it is another example of ambiguity in Wirth 1971, 1972. An accurate description of a language must say if this is a well-formed program or not.

In our semantics, no restriction is imposed on where the label may appear in the text. This means that jumps into (or out from) the body of a repetitive statement are allowed. The behavior of **segm** in such case will be described in their respective sections.

According to Wirth 1971 we do not allow jumps into a procedure body, but, contrary to Wirth 1972 we do not allow jumps out of a procedure activation, i.e. jumps cannot cause the change of the current frame. For this reason we have not introduced the label declaration statement of Wirth 1972 since the notion of scope for a label is meaningless to our semantics.

Lockhood Morris and others have suggested the notion of continuation as a possible way of defining the semantics of programming languages with the goto instruction. It cannot be used in LCF in a straightforward way since a type conflict arises. On the contrary in our semantics no type conflict is introduced by the goto, in fact its semantics simply reduces to changing the first argument of MS. The text to be executed next is replaced by the text evaluated by the **segm** function.

3.3.1.2 Assignment Statement

The semantics of the assignment statement is defined by the combinator ASSIGN:

```
ASSIGN = [∝F.[λn v f s.
        n=FUNV→ISADMISVAL(s(f,typeloc FUNV),v(s))→STORE(f,s,FUNV,v(s)),UU,
        ISINTYPE(n,v,f,s)→STORE(f,s,LOCOFVAR(n,f,s),v(s)),
        istopf(f)→UU,
        ISFUNFR(f,s,NEWFP(n,f,s))→F(VARBNDTO(n,f,s),v,NEWFP(n,f,s),s),UU]].
```

First of all a test is done to see whether the location to be assigned is FUNV, i.e. if we are assigning the value to a function identifier in a function activation (see 3.2.1.3). In this case if the typeloc FUNV is present in the current frame and the value v matches with its content, the combinator STORE stores v(s) in FUNV (see appendix 3.9). Otherwise ASSIGN returns the undefined store. If n is not FUNV, then the current frame is checked. If n has been declared in it and the value v matches with its type then the assignment takes place. A type mismatch makes the assignment to return the undefined store. If n is not local to the current frame, it may be a name parameter or a free variable for that frame. In both cases ASSIGN applies recursively with a mechanism quite similar to FETCHV (see 3.2.1.2). The only difference is that here a test is done by ISFUNFR to see if the assignment may cause a side effect in a function activation.

```
ISFUNFR = [∝F.[λf s nf. ISLOCAL(FUNV,s(f))→ FF,pred f=nf → TT,F(pred f,s,nf)]].
```

ISFUNFR checks if any frame between those pointed to by f and nf is a function frame, i.e. if FUNV is local to it.

The auxiliary combinator ISINTYPE:

```
ISINTYPE = [λv val f s.ISLOCAL(typeloc NAMOFVAR(v),s(f))→ISADMISVAL(TYPOFVAR(v,f,s),val(s)),FF].
```

evaluates to true if the variable v is local to the frame s(f) and the value val is compatible with its type. It evaluates to false if v is not local to s(f) and to undefined if a type mismatch occurs. The definition of the combinators used in ISINTYPE may be found in appendix 3.7,-9.

3.3.1.3 Procedure Statement

When a procedure is activated, its formal arguments are bound to the actual arguments in a new frame obtained by increasing the current frame pointer by 1. In such frame a location textloc is created whose content is the statement part of the activated procedure, and a location alnk is created containing the pointer to the frame where the procedure has been declared.

By looking at the definition of MS given in 3.3 we see that, when a procedure statement is executed, the auxiliary combinators PROCPAL, PROCBODY, PROCDECL are used. They are defined in appendix 3.8 and are used for fetching the formal argument list, the declaration part and the statement part of the activated procedure.

The set up of the new frame and the binding of the parameters is done by MPB:

```
MPB = [λfa aa f s n.BIND(fa,aa,succ f,MAKFRAME(PROCBODY(n,f,s),PFLNK(n,f,s),succ f,s))].
```

MAKFRAME sets up a new frame and creates the locations textloc and alnk in it. At the end of the procedure activation such frame is deleted by CLEAR:

```
CLEAR = [λf s f1.(f1=f)→UU,s(f1)].
```

CLEAR makes it explicit that the local variables of the procedure frame are no longer in the store.

The bindings of the parameters in a procedure activation is the same as that of a function activation. It is defined by:

```
BIND ≡ [∝F.[λfa aa f s.
        iseof fa → (iseof aa → s,UU),
        isparameter(fstof fa) →F(rmdof fa,rmdof aa,f,MKBINDING(fstof fa,fstof aa,f,s)),UU]].
```

Corresponding parameters in the two lists are bound by MKBINDING. If the two lists have different length the binding results in an undefined store. PASCAL allows procedures without parameters. In such case the abstract syntax for the two parameter lists is the empty list EOF.

The MKBINDING combinator is defined as:

```
MKBINDING ≡ [λfa aa f s.
      isvarp(fa) → TYMATCH(fa,typeloc,aa,f,s) →
                   CREALOC(f,s,bindloc,namof fa,EXPRFORV(aa)),UU,
      isvalp(fa) → ASSIGN(namof fa,MEXPR(aa,f),f,
                   CREAV(f,namof fa,typof fa,CRNTF(f,s),s)),
      isfunp(fa) → TYMATCH(fa,typfunloc,aa,f,s) →
                   CREAF(f,namof fa,FUNCDEF(aa,f,s),typof fa,CRNTF(f,s),PFLINK(aa,f,s),s),UU,
      isprocp(fa)→ CREAP(f,namof fa,PROCDEF(aa,f,s),PFLINK(aa,f,s),s),UU].
```

If the formal parameter fa is a variable parameter (i.e. a parameter passed by name) then, if its type matches the type of the actual parameter aa, a binding location bindloc (namof fa) is created. Its content is the EXPRFORV(aa). If aa has subscripts they must be evaluated when the binding takes place (see Wirth 1971). This evaluation is performed by EXPRFORV which substitutes a numeral for the value of each subscript.

The test on the type matching between formal and actual parameters is done by TYMATCH:

```
TYMATCH ≡ [λfa loc aa f s.TYPEVAL(typof fa,CRNTF(f,s),s)=TYPEDEF(loc aa,pred f,s)].
```

The type identifier associated with the formal argument is evaluated (by TYPEVAL) in the frame where the procedure has been declared. The pointer to it is retrieved by CRNTF. We have in fact chosen to evaluate the type associated with the formal arguments of a procedure when it is activated and not when it is declared. The type of the actual argument is fetched from the store by the TYPEDEF combinator in the location typeloc aa or typfunloc aa depending on whether fa is a variable or function parameter. All these auxiliary combinators are defined in appendix 3.8. Here we only describe TYPEVAL:

```
TYPEVAL ≡ [∝F.[λn f s.
      isbasetype n → n,
      isarspec  n → mkarspec(F(arlimof n,f,s),F(typelof n,f,s)),
      istyppart n → iseof n  → n,
                    ispair n → mkpair(F(fstof n,f,s),F(rmdof n,f,s)),UU,
      ISLOCAL(typeloc n,s(f))→F(s(f,typeloc n),f,s),
      istopf f → UU,F(n,CRNTF(f,s),s)]].
```

If the type n being evaluated is a base type, i.e. integer or subrange, then TYPEVAL evaluates to it. If

n is an array specification, then both the types of its subscripts and the type of its elements are recursively evaluated. The types of the subscripts of an array are given as a list of subranges. This list satisfies the predicate istyppart, so each one of its elements is recursively evaluated. Finally, if the type being evaluated is a type identifier defined in the current frame, then TYPEVAL applies recursively to its definition. If the type definition is not found in the current frame, then the appropriate frame is searched.

If a formal parameter fa is passed by value, then a variable fa is declared in the current frame by CREAV (see 3.1.2). Its type is evaluated by TYPEVAL in the appropriate frame and stored into the location typeloc fa. The value of the actual parameter aa is then computed by MEXPR and assigned to fa. ASSIGN checks whether or not the types of fa and aa are compatible (see 3.3.1.2).

If the formal parameter fa is a function parameter and the type of fa matches with that of aa, a function fa is declared in the current frame by the combinator CREAF (see 3.1.3). The type of this function is the type of fa evaluated by TYPEVAL in the appropriate frame. In its accink location the content of the accink location of aa is stored. The text of the actual argument is retrieved by FUNCDEF, its accink by PFLINK and its type is evaluated by TYPEVAL in the usual way.

If the formal parameter fa is a procedure parameter a procedure [illegible] is declared in the current frame by CREAP. In the accink of such procedure the content of the accink location of the actual parameter is stored.

Since the combinators used for binding formal and actual parameters are those used in declarations (see 3.1.2,-3), an undefined store is returned if the reserved identifier FUNV is used as formal parameter (see 3.2.1.3 for a discussion on the use of FUNV). From the definition of MKBINDING it is also evident that FUNV cannot be used as an actual parameter since both EXPRFORV and MEXPR return an undefined result if applied to FUNV. The auxiliary combinators used by MKBINDING test, by ISPRESENT, the presence of identifiers in a frame. It follows that an identifier cannot appear twice as formal parameter and in the declaration part of a procedure.

Procedures, as well as functions (see 3.2.1.3), cannot be executed recursively if they declare a procedure or have a formal procedure parameter with the same name.

As noted for functions, a procedure may also have itself as actual argument. Even though LCF is a typed logic, we avoid type conflicts by passing texts, and not functions as parameters.

3.3.1.4 Read Statement

PASCAL has no read and write statements. We have introduced them for defining the semantics of the input and output. In Wirth 1972 a standard procedures, *read* and *write*, are introduced for handling the input and output.

As said in 2.2 the data to be inputed is stored into the fileloc INP location of the store by the PASCAL function. Whenever the value of a variable has to be inputed, it is read from the buffer INP by the READ function:

```
READ = [λn f s.ISFUNFR(f,s,Ø)→ASSIGN(n,MEXPR(fstof(IBUFFER s),f),f,
                    STORE(Ø,s,fileloc INP,rmdof(IBUFFER s))),UU].
```

A test is done to see if the read statement is executed during a function activation, in this case the result of READ is undefined. Otherwise its result is a new store where the first element of the input buffer has been removed and its value has been assigned to the variable being read.

3.3.1.5 Write Statement

The results produced by a program are stored into the fileloc OUT location, where they are eventually retrieved by the OUTPUT combinator (see 2.2). The write statement puts into the buffer the numeral of the value of the variable to be outputed.

```
WRITE ≡ [λn f s.ISFUNFR(f,s,B)→STORE(B,s,fileloc OUT,mkpair(mknumconst(FETCHV(n,f,s)),
                                                    OBUFFER s)),UU]].
```

As with the read statement, it is forbidden to write during a function activation.

3.3.2 Structured Statements

The structured statements included in our version of PASCAL are:

1) the conditional statement in its two forms: *if-then* and *if-then-else*,
2) the repetition statements *while* and *repeat*,
3) the for statement in its two forms: *for-to* and *for-downto*.

We have not included the *case* and the *with* statements defined in Wirth 1971, 1972 since they do not seem very relevant to the integer arithmetic part of PASCAL. In Wirth 1971, 1972 the compound statement is also included in the list of structured statements. In our description of PASCAL the compound statement does not appear since the *begin*, *end* delimiters are not present in the abstract syntactic form of a program. The compound statement in its abstract syntactic form is a list of statements assembled by the syntactic constructor mkcmpnd and ending with the symbol ES. The semantics of the compound statement is defined by MS which establishes the flow of the control through the statement part of the program text.

3.3.2.1 Conditional Statement

The conditional statement in PASCAL has two forms: *if-then* and *if-then-else*. In the abstract syntactic form the conditional statement always has an else part, possibly it reduces to the empty statement ES.

The semantics of the conditional statement is defined by the combinator COND:

```
COND ≡ [λq f g s.(q(s)→f(s),g(s))].
```

The test of the conditional is evaluated in the store where the conditional statement is executed. The conditional returns the then-part or the else-part evaluated in this store, depending on the value of the test.

Going back to the definition of MS given in 3.3, we see that if the first statement of the text in execution is a conditional, its test is evaluated by the MBEXPR combinator and then MS applies

recursively to the text resulting from appending the then-part or the else-part of the conditional to the remaining statements. The **append** function, defined in appendix 2.5 corresponds to the ordinary appending function for lists.

If a goto statement is executed within a branch of a conditional, then the execution goes on with the text furnished by the **segm** function. If a jump into a branch of a conditional is done, then the text to be executed next consists of all the statements between the first occurrence of the label to jump to and the end of the branch of the conditional, appended to the rest of the program. This text is the result of the **segm** function defined in 3.3.1.1.

3.3.2.2 While and Repeat Statements

The while statement is a repetition statement whose abstract syntax is:

mkwhile(test,body).

body is repeatedly executed until **test** becomes false. The semantics of the while statement as given in MS (see 3.3) can be explained as follows: **test** is evaluated, if its result is true, then MS applies recursively to **body** appended to the while statement itself and to the remaining statements in execution. If the test fails, MS applies to the remaining statements.

Wirth 1971 says that in PASCAL, for all e and S the two statements

while e do S

and

if e then begin S; while e do S end

are equivalent. We prove this true for our semantics (see 4.4).

The repeat statement is similar to the while statement. The only difference is in that the repeat first executes its body and then performs the test to see whether to go on or stop. The semantics of the repeat statement is defined in MS (see 3.3). MS applies recursively to the body of the repeat, appended to a conditional (specifying whether or not the repeat must be executed again), appended to the remaining statements in execution.

We have also proved the equivalence described in Wirth 1971 for the repeat statement, i.e. for all e and S the two following statements are equivalent:

repeat S until e

and

begin S; if ¬e then repeat S until e end

In Weyhrauch and Milner 1972 and in Aiello and Aiello 1974 a **WHILE** combinator was introduced for defining the semantics of the while statement:

WHILE ≡ [∝F.[λt b.COND(t,b⊗F(t,b),ID)]].

It cannot be used here since a goto statement can stop the execution of the body of the while. We can prove that the definition of the semantics for the while statement given in MS reduces to the above semantic combinator when the body of the while is goto free (see 4.3).

The language described in Weyhrauch, Milner 1972 had no repeat statement. The semantics for the repeat statement was described in Aiello, Aiello 1974 by the combinator REPEAT:

REPEAT ≡ [∝F.[λb t. b⊗COND(t,F(b,t),ID)]].

It is similar to the WHILE combinator described above and the same considerations concerning the presence of goto's hold for it.

If a goto statement is executed within the body of a while or repeat statement, then the execution of the repetition statement is stopped and the text to be executed next is furnished by the **segm** combinator. From the definition of **segm** given in 3.3.1, we see that when a goto statement jumps into the body of a repeat (while) statement the piece of body starting from the first occurrence of the label is appended to the text starting from that repeat (while) statement. This means that the part of body from the label to the end is executed and then a test is done to see whether or not the execution of the repetition statement must be stopped or goes on.

3.3.2.3 For Statement

In PASCAL the for statement has two forms:

for i:=e1 to e2 do b;

and

for i:=e1 downto e2 do b;

In both cases *b* is the body of statements which is repeatedly executed, and *i* is the variable which controls the loop. In the for-to statement it is increased by 1 each time *b* is executed. In the for-downto statement it is decremented by 1. The two expressions *e1* and *e2* will be referred to as the initial and final values of the control variable.

The abstract syntax for the two forms of for statements is defined by:

mkforto(i,e1,e2,b),

mkfordn(i,e1,e2,b).

Their semantics is defined in MS. A test is done to check if the value of the control variable i is equal to the final value **e2**. The test is:

fortest ≡ [λx .isforto(x)→mkrel(lseq,lbof(x),ubof(x)),isfordn(x)→mkrel(greq,ubof(x),lbof(x)),UU].

If **fortest** evaluates to **TT**, the initial value **e1** is assigned to the control variable i, then the meaning

function MS applies to the body of the for statement appended to the text assembled by the combinator fortoup (fordnup):

```
fortoup = [λx .mkcmpnd(mkforto(indexof(fstof(x)),mkexpr1(plus1,indexof(fstof(x))),
                       ubof(fstof(x)),bodyof(fstof(x))),rmdof(x))],

fordnup = [λx .mkcmpnd(mkfordn(indexof(fstof(x)),mkexpr1(minus1,indexof(fstof(x))),
                       lbof(fstof(x)),bodyof(fstof(x))),rmdof(x))].
```

fortoup (fordnup) updates the initial value of the for loop by substituting i+1 (i-1) for i.

We have chosen to define the for in terms of the algorithmic equivalences given in Wirth 1971, i.e. for all *i*, *e1*, *e2* and *S* the statement:

for i:=e1 to e2 do S

is equivalent to

```
if e1≤e2 then
begin i:=e1;S;
   for i:=succ(i) to e2 do S
end
```

and the statement

for i:=e1 downto e2 do S

is equivalent to

```
if e1≥e2 then
begin i:=e1;S;
   for i:=pred(i) to e2 do S
end
```

We have imposed no restrictions on the fact that the values of *i*, *e1* and *e2* are changed by *S* or by the for statement itself, or on the jumps into or out from the body of a for statement. The value of the control variable at completion of the for has the last value assumed, namely the value it had after the last execution of *S*. This interpretation of the for statement is different from the description of the PASCAL for statement as given in Wirth 1971, 1972 and in Hoare and Wirth 1973. The definitions given in these three papers are indeed different from each other. Our choice has been motivated by the fact that we wanted the semantics of the for statement to be as smooth as possible and, at the same time, we wanted to make it less ambiguous then Wirth 1972. The definition of the for, given in terms of the above algorithmic equivalences in Wirth 1971, was changed in Wirth 1972, following the suggestions made in Hoare 1972. In order to leave the implementer more freedom, the following equivalences are required in Wirth 1972:

for i:=e1 to e2 do S

is equivalent to

i:=e1; S; i:=succ(i); S; ... i:=e2; S

and

***for** i:=e1 **downto** e2 **do** S*

is equivalent to

i:=e1; S; i:=pred(i); S; ... i:=e2; S

These definitions seem ambiguous to us: what happens if *e1>e2* in the for-to statement?

The third definition of the PASCAL for statement is given in Hoare and Wirth 1973. This is closer to that given in Wirth 1972, but not the same. It is given in axiomatic form:

$$\frac{(a \leq x \leq b) \wedge P([a..x))\ \{S\}\ P([a..x])}{P([])\ \{\textbf{for}\ x:=a\ \textbf{to}\ b\ \textbf{do}\ S\}\ P([a..b])}$$

$$\frac{(a \leq x \leq b) \wedge P((x..b])\ \{S\}\ P([x..b])}{P([])\ \{\textbf{for}\ x:=b\ \textbf{downto}\ a\ \textbf{do}\ S\}\ P([a..b])}$$

It is written in the formalism proposed by Hoare 1969, where P{Q}R means that if P and R are predicates and P is true before the execution of the body of statements Q, and Q terminates, then R is true after the execution of Q. [a,b] denotes the interval {x|a≤x≤b}, [a,b) denotes the interval {x|a≤x<b}, and so on. This rule was used in Hoare 1972 for characterizing the correctness of the for statement. Apart from the fact that the description of the rule given in Hoare 1972 and that given in Hoare and Wirth 1973 are different, we do not agree with it. In fact it leaves unspecified what happens when the for-to statement is executed with the initial value greater then the final value. It seems to us that any definition which leaves this ambiguous cannot serve as a satisfactory specification of the meaning of the for statement. In particular it cannot be used to prove general theorems about the for statement. Consider for example an implementation of PASCAL in which if *b<a* in one of the above for statements, then the body of statements *S* is executed 14 times! This implementation satisfies the above axioms, but is certainly strange.

In Wirth 1971, 1972 nothing is said about the behavior of the goto's with respect to the for statement. Hoare and Wirth 1973 do not deal with goto's. In our semantic definition, if a goto statement is executed within the body of a for statement, then the execution of the repetition statement is stopped and the text returned by **segm** is executed next. From the definition of **segm** we see that if a jump into the body of a for statement is executed, then **segm** returns the piece of body starting from the first occurrence of the label to jump to, appended to the piece of abstract syntax returned by the **fortoup** or **fordnup** combinators.

If a jump into the body of a for statement is executed we distinguish between two cases: 1) the jump is from one point to another point of the body of the same for statement. In this case the computation goes on with the control variable having the current value. 2) the jump is from a point of the program outside the for statement. In such case the computation may result in the undefined

store accordingly to whether or not the control variable has been assigned a value prior to the execution of the jump. In fact the updating combinators fortoup and fordnup replace i+1 and i-1 for e1 in the for statement, so it evaluates to UU if the control variable has not yet been assigned a value.

Haberman 1973 dislikes the possibility of jumping into a for statement. We have allowed such jumps, thus a for loop may be initialized from outside and started by a jump. This seems reasonable since PASCAL has no block structure, so the control variable of a for statement has to be declared in the declaration part of the text and may be given a value independently of the for statement. Furthermore, since the control variable is not local to the for statement, we do not see any reason for leaving it undefined after the execution of the for statement, as required in Wirth 1972. Nothing is said at this regard in Wirth 1971 and in Hoare and Wirth 1973. We do not agree that a perfectly behaved statement should leave an undefined value in a location which has been declared and assigned a value. It also leaves ambiguous what happens to the control variable if a goto stops the execution of the for loop.

Our semantics doesn't check to see if the control variable, the initial value or the final value are modified during the execution of the for statement. This makes our for statement similar to the while statement. Since the control variable is not a dummy variable of the loop there is no reason for it to be treated differently from any other variable. Wirth 1971, 1972 and Hoare and Wirth 1973 are discordant about the requirements on such modifications. Moreover it is our opinion that checking for them is very difficult and is unlikely to be done in any current implementations of PASCAL. Consider for example a program where an integer variable *i* is declared which also declares the following procedure:

```
procedure A(j,k:integer)
for i:=j to k do
    if i=3 then A(k+1,j)
           else A(j+1,k);
```

Note that in this program the control variable is changed by the recursion of the procedure *A*, not by an assignment statement.

A final point regarding our semantics: as with the while and repeat statements, if a text is goto-free the semantics of the for statement can be defined by the following two combinators:

```
FORTO = [∝F.[λi e1 e2 b f. COND(MBEXPR(mkrel(leq,e1,e2),f),ASSIGN(i,MEXPR(e1,f),f)⊗b⊗
                    F(i,mkexpr1(plus1,i),e2,b,f),ID)]];

FORDN = [∝F.[λi e1 e2 b f. COND(MBEXPR(mkrel(geq,e1,e2),f),ASSIGN(i,MEXPR(e1,f),f)⊗b⊗
                    F(i,mkexpr1(minus1,i),e2,b,f),ID)]];
```

The equivalence, in the goto-free case, between the definition of the semantics of the for statement given in MS and that given by the two above combinators, can be proved easily (see 4.3).

SECTION 4 PROPERTIES OF THE SEMANTICS

In this section we discuss some general properties of the interpretation of PASCAL in LCF. We have proved :

1) the meaning function MS is strict on the store, i.e. for any statement st and any framepointer f, MS(st,f,UU)≡UU.

2) for goto-free programs, MS is a homomorphism with respect to the constructor mkcmpnd, i.e. ∀f.MS(mkcmpnd(a,b),f)≡MS(a,f)⊗MS(b,f).

3) MS reduces to a simpler function for goto-free programs. New combinators defining the semantics of the repetition statements are given.

4) all the equivalences about repetition statements given in Wirth 1971 hold in our semantics.

5) some miscellaneous theorems about MDEC, MDEF, MS

Section 4.1 The strictness of MS on the store

The main theorem of this section is

∀st f.MS(st,f,UU)≡UU.

We do not show the proof here as it is a single LCF simplification using the lemma

∀t a b.(t→a,b)(UU)≡(t→a(UU),b(UU))

The main theorem should not be regarded as trivial however, as it requires 208 substitutions. Without the LCF simplifier, this proof would have been over 1000 steps long. This is an important theorem because it shows that our interpretation of statements behaves correctly with respect to the termination of computations.

Consider the following program

```
   var n:integer
   begin
1: goto 1;
   n:=1;
   end
```

This program fails to terminate. To us it seems that the only reasonable interpretation of this program must be the undefined function. If the meaning function is not strict, it may happen that the assignment of 1 to *n* builds up a store in which *n* has value 1. Suppose we were to choose the most obvious interpretation of assignment, i.e. if the above program is being executed in a store **s**, and a frame whose framepointer is f then the meaning of the assignment statement in the example is a new store **s1**:

```
s1 = [λfr.fr=f→[λm.m=n→1,s(f,m)],s(fr,m)],
```

so

```
s1(f) = [λm.m=n→1,s(f,m)].
```

This new store has the unfortunate property that even if s=UU, we still have s1(f,n)=1. It is thus not undefined.

The desire for the interpretation of a program to be an extensionally given function on the store and composition of these functions to correspond to executing one program after another, means that an interpretation which is strict on the store is the only one that makes sense. In Hoare's axiomatic treatment this problem goes away but the price is that every statement that you can prove about a program is conditional on its termination. In the above case one proves the sentence, "If the program terminates then n=1".

Because, as already said, the proof is a single step we do not give it here. Instead we will explain why for our semantics ASSIGN is strict on the store. The "***" represent some arbitrary combinator.

```
ASSIGN = [λn v f s.n=FUNV→ISADMISVAL(s(f,typeloc FUNV),v(s))→ ***,ISINTYPE(n,v,f,s)→ ***,***]
```

So

```
ASSIGN(n,v,f,UU) = n=FUNV→ISADMISVAL(UU,v(UU))→ ***,UU,ISINTYPE(n,v,f,UU)→ ***,***]
```

ISADMISVAL asks if a value is of an admissible type. UU is not even a type, no less admissible, so ISADMISVAL returns UU.

```
ISINTYPE(v,val,f,UU)=ISLOCAL(typeloc NAMOFVAR(v),UU))→ISADMISVAL(TYPOFVAR(v,f,UU),val(UU)),FF]

ISLOCAL(loc,UU) = UU=UNDEF→FF,TT
```

But for any X, UU=X is just UU so ISLOCAL(loc,UU)=UU. This is the central point of the entire strictness proof. Looking up a location in a defined store in an existing frame is not undefined if that location has not been created. Stores are constructed in such a way that we can test if it is defined and no assignment is made if it isn't. This check is done by ISLOCAL, which returns UU if the frame is undefined. The proof is completed by making the correct substitutions.

Other theorems about strictness appear in section 4.5.

Section 4.2 Properties of MS for goto-free programs

A goto-free program is defined by the following predicate :

```
isgotofree = [∝F.[λ s .
              isgoto    s →  FF,
              issingle  s →  TT,
              islabstat s → F(statmof s),
              isiter    s → F(bodyof s),
```

```
        iscond   s → F(thenof s) ∧ F(elseof s),
        iscmpnd  s → F(fstof s) ∧ F(rmdof s),UU]],
```

where **issingle** and **isiter** are predicates satisfied by the simple and the repetition statements respectively (see appendix 2.4). The main theorem about goto-free programs is:

```
∀ S P f.isgotofree(S)::isgotofree(P):: MS(append(S,P),f) ≡ MS(S,f) ● MS(P,f).
```

It states that if S and P are texts without any goto statement, then the result of the application of MS to the concatenation of them is the same as the functional composition of the application of MS to each of them. The proof of this theorem is based on a case analysis on the first element of S. We have not included it in the paper as it is rather long even if very simple. We didn't find any proof by induction on **isgotofree**, so we proved it by induction on MS. To do this the two following lemmas are to be proved:

```
∀S P f.isgotofree(S)::isgotofree(P)::MS(append(S,P),f)⊂MS(S,f)⊗MS(P,f)
∀S P f.isgotofree(S)::isgotofree(P)::MS(S,f)⊗MS(P,f)⊂MS(append(S,P),f)
```

In section 4.1 it has been noted that the proof of the strictness of MS on the store depends on a theorem about conditional expressions. For proving the above lemmas with a similar proof we needed the following theorem about conditional expressions:

```
∀t.(t→a,b) ⊂ (t→d,f)     ASSUME a⊂d, b⊂f.
```

Unfortunately the current version of the LCF conditional simplifier doesn't handle sentences of the form A⊂B as simplification rules, even though in this case no specific property of the symbol ⊂ is involved.

The above homomorphism theorem is analogous to the Hoare's composition rule for statements, valid for goto-free programs. This theorem, as well as Hoare's rule is not valid in general. Consider the following example:

```
    a:=1;
    goto 1;
    a:=3;
1:  a:=a+1;
```

the corresponding abstract syntax is:

```
P ≡ mkcmpnd(mkass(a,mknumconst 1),
      mkcmpnd(mkgoto 1),
        mkcmpnd(mkass(a,mknumconst 3),
          mkcmpnd(mklabstat(1,mkass(a,mkbexpr1(plus1,a)),ES))))
```

The meaning of this program in the frame specified by the frame pointer f is defined by MS(P,f). The validity of the composition rule would imply the following equivalence:

```
MS(P,f) ≡ MS(mkcmpnd(mkass(a,mknumconst 1),ES),f)⊗
          MS(mkcmpnd(mkgoto(1),ES),f)⊗
            MS(mkcmpnd(mkass(a,mknumconst 3),ES),f)⊗
```

MS(mkcmpnd(mklabstat(1,mkass(a,mkbexpr1(plus1,a))),ES),f),

which is false: starting from a store where **a** is declared in the current frame, MS(P,f) returns a store where, in the current frame, **a** has value 2, while the right hand side evaluates to a store where, in the current frame, **a** as value 4. The right hand side is wrong, since by interpreting each statement separately, it is impossible to skip a piece of text as required by a goto.

In the next section we consider how the semantics of a PASCAL statement part is simplified when it is goto-free. Our semantics deals also with programs where the composition rule is not valid. Hoare axiomatic approach to the definition of the semantics of a programming language relies on the validity of the composition rule, so it cannot easily treat programs with goto's. Hoare and Wirth 1973 axiomatization of PASCAL, for instance, doesn't define the goto statement. The Igarashi, London and Luckham 1973 VCGEN, based on this approach, deals only with backwards goto's and preserves the validity of the composition rule by considering indivisible the piece of program between the label to jump to and the goto.

Section 4.3 An equivalent meaning function for goto-free programs

As noted in the description of repetition statements (see 3.3.2.2,-3), if the body of the repetition statement is goto-free, new combinators may be defined for describing their semantics. In this case the semantics defined by MS is the same as that defined by the new combinators.

The proofs of the first four equivalences are quite similar; they are carried out by subgoaling to the two goals with the logical symbols $\supset$, $\subset$ respectively. All these proofs are standard and could be automated by enriching the features of the current LCF system. In appendices 4,5,6 we have included the commands and the printouts of the proof of one half of each of the first three equivalences. The fourth is analogous to the third one.

The proof of the equivalence between MS and MSGTFR is carried out by proving the lemmas with $\subset$, $\supset$ respectively, and using the above equivalences for repetition statements. A long case analysis on **S** is performed, analogous to that discussed in 4.2. Even in this case the proof could become very short by improving slightly the LCF conditional simplifier.

1) ∀S t f .isgotofree(S):: MS(mkcmpnd(mkwhile(t,S),ES),f) = WHILE(MBEXPR(t,f),MS(S,f))

where WHILE = [∝F.[λt b.COND(t,b⊗F(t,b),ID)]]

2) ∀S t f. isgotofree(S):: MS(mkcmpnd(mkrepeat(S,t),ES),f) = REPEAT(MS(S,f),MBEXPR(mkbexpr1(not,t),f))

where REPEAT = [∝F.[λb t.b⊗COND(t,F(b,t),ID)]]

3) ∀S i e1 e2 f .isgotofree(S):: MS(mkcmpnd(mkforto(i,e1,e2,S),ES),f) = FORTO(i,e1,e2,MS(S,f),f)

where FORTO = [∝F.[λi e1 e2 b f. COND(MBEXPR(mkrel(leq,e1,e2),f),ASSIGN(i,MEXPR(e1,f),f)
⊗b⊗F(i,mkexpr1(plus1,i),e2,b,f),ID)]];

4) ∀S i e1 e2 f .isgotofree(S):: MS(mkcmpnd(mkfordn(i,e1,e2,S),ES),f) = FORDN(i,e1,e2,MS(S,f),f)

where FORDN = [∝F.[λi e1 e2 b f. COND(MBEXPR(mkrel(geq,e1,e2),f),ASSIGN(i,MEXPR(e1,f),f)

```
                    ⊗b⊗F(i,mkexpr1(minus1,i),e2,b,f),ID)]];

5) ∀S f. isgotofree(S):: MS(S,f) = MSGTFR(S,f)

    where
MSGTFR=[∝F.[λst f.
      isemptyst st    → ID,
      iscmpnd st      →
        isemptyst(fstof st)→ F(rmdof st,f),
        islabstat(fstof st)→ F(statmof(fstof st),f)⊗F(rmdof st,f),
        isread(fstof st)   → READ(namof(fstof st),f)⊗F(rmdof st,f),
        iswrite(fstof st)  → WRITE(namof(fstof st),f)⊗F(rmdof st,f),
        isass (fstof st)   → ASSIGN(lhsof(fstof st),MEXPR(rhsof(fstof st),f),f)⊗F(rmdof st,f),
        isproccall(fstof st)→[λs.MPB(PROCFAL(namof(fstof st),f,s),
                               actargof(fstof st),f,s,namof(fstof st))]⊗
                             [λs.MD(PROCDECL(namof(fstof st),f,s),succ f,s)]⊗
                             [λs.F(PROCBODY(namof(fstof st),f,s),succ f,s)]⊗CLEAR(succ f)⊗F(rmdof st,f),
        iscond(fstof st)   → COND(MBEXPR(testof(fstof st),f),
                             F(thenof(fstof st),f),F(elseof(fstof st),f))⊗F(rmdof st,f),
        iswhile(fstof st)  → WHILE(MBEXPR(testof(fstof st),f),F(bodyof(fstof st),f))⊗F(rmdof st,f),
        isrepeat(fstof st) → REPEAT(bodyof(fstof st),MEXPR(mkboxpr1(not,testof(fstof st)),f))⊗F(rmdof
        isforto(fstof st)  → FORTO(indexof(fstof st),lbof(fstof st),
                             ubof(fstof st),bodyof(fstof st),f)⊗F(rmdof st,f),
        isfordn(fstof st)  → FORDN(indexof(fstof st),ubof(fstof st),
                             lbof(fstof st),bodyof(fstof st),f)⊗F(rmdof st,f),UU,UU]]
```

The definition of MSGTFR shows how our semantics simplifies for goto-free programs. No manipulation of the text is required, every statement can be treated independently of the others, some combinators as fortest, fortoup, fordnup, append are no longer necessary. The semantic combinators for repetition statements not only simplify the form of MS but also the proofs of properties of goto-free programs. In fact, in the general case proofs by induction on the repetition statement must be done by inducting on MS. For goto-free programs the induction can be directly done on the appropriate semantic combinator. Hence, only properties of the body of the repetition statements and not of the whole program are involved. The structure of the program reflects directly on the structure of the proof since allows to factorize it into easier lemmas.

In section 5.1 two different programs which compute the factorial function are compared. In the first one the iteration is performed by a while statement, in the second one by a backwards goto. The proofs of their correctness are different, the goto-free case is more straightforward. The proof of the correctness of the goto program may be reduced to that of the goto-free program by showing that, in general, a while loop is equivalent to an appropriate loop controlled by a conditional goto. This example shows the advantage of a formalism which allows to prove general properties of the language and the necessity of creating the right environment of theorems about the programming language to greately simplify the proofs of properties of programs.

Section 4.4 Equivalences for repetitive statements

In giving an interpretation of PASCAL in LCF our aim was to be as close as possible to the informal description given in Wirth 1971. For this reason we proved most of the properties of the statements that are mentioned in that paper. The LCF theorems stating the equivalences for repetition statements given in Wirth 1971 are:

```
∀e S. MS(mkcmpnd(mkwhile(e,S),ES)) =
      MS(mkcmpnd(mkcond(e,append(S,mkcmpnd(mkwhile(e,S),ES)),ES),ES)),

∀e S f. MS(mkcmpnd(mkrepeat(S,e),ES),f) =
      MS(append(S,mkcmpnd(mkcond(mkbexpr1(not,e),mkcmpnd(mkrepeat(S,e),ES),ES),ES)),f),

∀i e1 e2 S f. MS(mkcmpnd(mkforto(i,e1,e2,S),ES),f) =
      MS(mkcmpnd(mkcond(mkrel(lseq,e1,e2),
        mkcmpnd(mkass(i,e1),append(S,mkcmpnd(mkforto(i,mkexpr1(plus1,i),e2,S),ES))),ES),ES),f),

∀i e1 e2 S f. MS(mkcmpnd(mkfordn(i,e1,e2,S),ES),f) =
      MS(mkcmpnd(mkcond(mkrel(greq,e1,e2),
        mkcmpnd(mkass(i,e1),append(S,mkcmpnd(mkfordn(i,mkexpr1(minus1,i),e2,S),ES))),ES),ES),f),
```

All the proofs of the above statements are one step proofs. In fact, we have defined the semantics of the repetition statements directly in terms of the equivalence described in Wirth 1971.

Section 4.5 Miscellaneous theorems on MDEC, MDEF, MS

Our aim in this section is not to give an exhaustive list of the properties of PASCAL, but rather to show some typical example of theorems which have been used in the proofs presented in this report.

First of all we want to state that type definitions and declarations are independent of the order. The theorem proved for type definitions is:

```
∀n1 n2 ty1 ty2 f s .
isname(n1)::isname(n2)::n1≠n2::ISABSENT(n1,s(f))::ISABSENT(n2,s(f)) ::
MDEF(mkcmpnd(mktypedef(n1,ty1),mkcmpnd(mktypedef(n2,ty2),ES)),f,s) =
MDEF(mkcmpnd(mktypedef(n2,ty2),mkcmpnd(mktypedef(n1,ty1),ES)),f,s);
```

This theorem states that if n1 and n2 are different names and they do not appear in the store, then the order of type definitions using these names as type identifiers is irrelevant. The predicates appearing in it have an obvious meaning: ≠ is the negation of =, ISABSENT is the negation of ISPRESENT. The proof of this theorem has not been included in the report since it is a very simple proof done by simplification and using some properties of conditional expressions. Analogously the following theorems can be proved. They state that declarations are independent of the order.

```
∀n1 n2 ty1 ty2 f s .
isname(n1)::isname(n2)::n1≠n2 ::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkvardecl(n1,ty1),mkcmpnd(mkvardecl(n2,ty2),ES)),f,s) =
MDEC(mkcmpnd(mkvardecl(n2,ty2),mkcmpnd(mkvardecl(n1,ty1),ES)),f,s);

∀n1 n2 ty1 ty2 fs2 f s .
isname(n1)::isname(n2)::n1≠n2::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkvardecl(n1,ty1),mkcmpnd(mkfundecl(n2,fs2,ty2),ES)),f,s) =
MDEC(mkcmpnd(mkfundecl(n2,fs2,ty2),mkcmpnd(mkvardecl(n1,ty1),ES)),f,s);

∀n1 n2 ty1 ty2 fs1 fs2 f s .
isname(n1)::isname(n2)::n1≠n2::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkfundecl(n1,fs1,ty1),mkcmpnd(mkfundecl(n2,fs2,ty2),ES)),f,s)=
MDEC(mkcmpnd(mkfundecl(n2,fs2,ty2),mkcmpnd(mkfundecl(n1,fs1,ty1),ES)),f,s);
```

```
∀n1 n2 ty1 fs1 ps2 f s .
isname(n1)::isname(n2)::n1≠n2::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkfundecl(n1,fs1,ty1),mkcmpnd(mkprocdecl(n2,ps2),ES)),f,s) =
MDEC(mkcmpnd(mkprocdecl(n2,ps2),mkcmpnd(mkfundecl(n1,fs1,ty1),ES)),f,s);

∀n1 n2 ty1 ps2 f s .
isname(n1)::isname(n2)::n1≠n2::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkvardecl(n1,ty1),mkcmpnd(mkprocdecl(n2,ps2),ES)),f,s) =
MDEC(mkcmpnd(mkprocdecl(n2,ps2),mkcmpnd(mkvardecl(n1,ty1),ES)),f,s);

∀n1 n2 ps1 ps2 f s .
isname(n1)::isname(n2)::n1≠n2 ::ISABSENT(n1,s(f))::ISABSENT(n2,s(f))::
MDEC(mkcmpnd(mkprocdecl(n1,ps1),mkcmpnd(mkprocdecl(n2,ps2),ES)),f,s) =
MDEC(mkcmpnd(mkprocdecl(n2,ps2),mkcmpnd(mkprocdecl(n1,ps1),ES)),f,s);
```

Some theorems describing properties of MDEF and MDEC are now listed. Each of them has been proved in one step.

```
∀ x y f. MDEF(mkcmpnd(x,y),f)=MDEF(x,f)⊗MDEF(y,f);

∀ x y f. MDEF(mkvardecl(x,y),f)= ID;

∀ x y z f.MDEF(mkfundecl(x,y,z),f)= ID;

∀ x y f.  MDEF(mkprocdecl(x,y),f)= ID;

∀ x y f. MDEF(mktypedef(x,y),f)= CREAT(f,x,y);

∀f.      MDEF(ES,f)=ID;


∀ x y f.  MDEC(mkcmpnd(x,y),f)=MDEC(x,f)⊗MDEC(y,f);

∀ x y f.  MDEC(mkvardecl(x,y),f)= CREAV(f,x,y,f);

∀ x y z f.MDEC(mkfundecl(x,y,z),f)= CREAF(f,x,y,z,f,f);

∀ x y f.  MDEC(mkprocdecl(x,y),f)= CREAP(f,x,y,f);

∀f.       MDEC(ES,f)=ID;
```

In the following we present some of the theorems dealing with MS, the combinators defining the semantics of statements and some predicates used by the semantic combinators. The proofs of these theorems are very simple (one step), however they were useful in proving programs as well as properties of MS.

```
∀f MS(ES,f)=ID;

∀x y f.MS(mkcmpnd(mkread x,y),f)=READ(x,f)⊗MS(y,f);

∀x y f.MS(mkcmpnd(mkwrite x,y),f)=WRITE(x,f)⊗MS(y,f);

∀x1 x2 y f.MS(mkcmpnd(mkass(x1,x2),y),f)=ASSIGN(x1,MEXPR(x2,f),f)⊗MS(y,f);
```

```
∀n f s.ASSIGN(n,UU,f,s)≡UU;

∀n e f.ASSIGN(n,e,f,UU)≡UU;

∀n f.WRITE(n,f,UU)≡UU;

∀n f.READ(n,f,UU)≡UU;

MEXPR(UU)≡UU;

BIND(UU)≡UU;

MPB(UU)≡UU;

∀l f.FETCH(l,f,UU)≡UU;

∀n f.PROCDEF(n,f,UU)≡UU;

∀n f.PROCFAL(n,f,UU)≡UU;

MD(UU)≡UU;

∀n f.PROCTXT(n,f,UU)≡UU;

∀n f.PROCDECL(n,f,UU)≡UU;

∀f.CLEAR(f,UU)≡UU;

∀loc.ISLOCAL(loc,UU)≡UU;

ISINBOUND(UU)≡UU;

∀ty.ISADMISVAL(ty,UU)≡UU;

∀v f s.ISINTYPE(v,UU,f,s)≡UU;

∀p e f.ISINTYPE(v,e,f,UU)≡UU;

∀f.ISPROCFRAME(f,UU)≡UU;
```

SECTION 5 EXAMPLES

In this section we want to discuss how to prove PASCAL programs in LCF. Two examples will be fully described:

1) the factorial program,

2) the McCarthy Airline reservation system.

We have also proved correct a PASCAL program for the computation of the GCD of two positive integers with the euclidean algorithm and a PASCAL program for the computation of the norm of a vector. These proofs have been executed using an earlier version of the LCF axiomatization of PASCAL and are described in Aiello and Aiello 1974. We have not rerun them on the final version of the axioms because, even though many details have been changed, the underlying ideas have not been modified, so the proofs would remain very similar.

Section 5.1 The factorial program

The partial correctness of a program for the computation of the factorial function has been already proved in LCF and discussed in Weyhrauch and Milner 1972. The proof presented here is very similar to that one. We have included it because the factorial program is a very simple and familiar example, so it is easy to go through the proof of its correctness. By comparing the proof given here and that given in Weyhrauch and Milner 1972 it may be seen that even though the programming language described here is much richer, the proof isn't more complex.

A PASCAL program which computes the factorial function is the following:

```
var n1,n2: integer
begin
read(n1);
read(n2);
while n2≠0 do
   begin n1:=n1*n2;n2:=n2-1; end;
write(n1);
end;
```

If the input consists of two nonnegative integers x and n this program computes $x*n!$. The factorial function is obtained if x equals 1.

In this program the repetition is performed by a while statement, hence we will call it while-program. An analogous program for the computation of the factorial function may be also written using a goto statement (it will be called goto-program):

```
var n1,n2: integer
begin
read(n1);
read(n2);
l: if n2≠0 then
```

```
    begin n1:=n1*n2;n2:=n2-1;goto 1; end;
  write(n1);
  end;
```

In LCF both programs are provable correct with respect to the function FACT:

```
FACT = [∝F.[λn x.n=0 → x,F(pred n,n*x)]],
```

FACT applies to two arguments n and x and evaluates to x*n!.

In the following, the LCF proof of the while-program is described in details. This program has no goto's, so the theorems described in 4.2 for goto-free programs can be used, making the proof much simpler. The proof of the second form of factorial program will only be sketched.

The abstract syntactic form of the while-program is:

```
FACTORIAL = mktext(DP,SP),

DP = mkcmpnd(mkvardecl(n1,INT),mkcmpnd(mkvardecl(n2,INT),ES)),

SP = mkcmpnd(mkread(n2),mkcmpnd(mkread(n1),
       mkcmpnd(mkwhile(test,body),mkcmpnd(mkwrite(n1),ES)))),

test = mkbexpr1(not,mkrel(eq,n2,mknumconst(0))),

body = mkcmpnd(mkass(n1,mkexpr2(times,n1,n2)),mkcmpnd(mkass(n2,mkexpr1(minus1,n2)),ES)).
```

The form of the LCF theorem to be proved is:

```
∀n x.isnat(n)::isnat(x)::APPLY(FACTORIAL,n,x)⊂FACT(n,x).
```

Informally, it says that the evaluation of the program FACTORIAL on the data n and x, if it terminates, gives the same result as the computation of the function FACT on n and x. APPLY is the following combinator:

```
APPLY = [λ p x y.fstof(FUNCT(p,EOF,LIST(x,y)))],

LIST  = [λ x y. mkpair(x,mkpair(y,EOF))].
```

As said in section 2, FUNCT maps sequences of integers into sequences of integers. Given a program p and two input numbers x and y, APPLY applies the combinator FUNCT to the sequence LIST(x,y) and then takes the first element of the output sequence.

The method used to prove the partial correctness of the while-program is quite standard for proving programs with a while loop. All the combinators appearing on the term at the left hand side are substituted by their definition. After some simplification (automatically done by LCF) the goal to be proved is:

```
∀n x . isnat(n) :: isnat(x) ::
 RESULT(WRITE(n1,Ø,WHILE(MBEXPR(test,Ø),MS(body,Ø),READ(n1,Ø,READ(n2,Ø,
 CREAV(Ø,n2,INT,Ø,CREAV(Ø,n1,INT,Ø,FRAMEØ(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) ⊂ FACT(n,x).
```

where RESULT is defined as ∀x.RESULT x ≡ fstof(OUTPUT x). The theorem on the while statement given in section 4.3 for goto-free programs has been used in achieving the above goal. The semantics of the loop is expressed in terms of the WHILE combinator. As it can be seen from the printout in appendix 7.2 the proof is done by induction on the WHILE combinator. The base case is trivially proved. The induction step is proved by cases on the predicate which controls the loop, i.e. ¬(n=Ø). If ¬(n=Ø) is false then the result easily follows, if ¬(n=Ø) is undefined a contradiction arises because n is a natural number. If ¬(n=Ø) is true, the goal is proved by a proper instantiation of the induction hypothesis. It is instantiated for **pred** n and x*n. Usually, in programs for the computation of the factorial of a natural number the variable n1 is not inputed a value, but it is initialized to 1. The initialization of n1 to x results in a strengthening of the induction hypothesis. In fact the variable x appears universally quantified in the statement of the theorem to be proved and can be properly instantiated. Actually the proved theorem is stronger than the desired one. The factorial program is obtained by giving the value 1 to x in the above theorem.

The proof given in appendix 7.2 is generated by the list of commands given in appendix 7.1. We want again to point out that LCF is not an automatic theorem prover. It has only a subgoaling mechanism and a sophisticated simplification algorithm which converts terms and simplifies them by using the axioms and theorems put (by the user) into a "simplification set".

In the simplification set there are all the syntactic constructors and selectors, plus the semantic combinators appearing in the first line of the list of commands. Note that LCF labels are prefixed by a ".", each axiom has been labeled with an identifier equal to the combinator being defined, and INDUCT is the label of the induction hypothesis. The modifications done to the simplification set after the proof is started (SS+/-something) are done only to increase the readability of the goals. In addition, to increase the readability of the proof, a combinator FRAME1 is introduced to describe an intermediate store:

```
FRAME1 ≡ [λf n x.[λf.f=0→
                  [λloc.loc=n2 →n,
                       loc=n1 →x,
                       loc=typeloc n2 → INT,
                       loc=typeloc n1 → INT,
                       loc=fileloc INP→ EOF,
                       loc=fileloc OUT→ EOF,
                       loc=textloc    → t,UNDEF],UU]].
```

In the printout of the proof each step appears with its "reason", namely the tactic used in achieving it, as well as the step numbers of the axioms and the names of the theorems involved in the simplifications. The theorems TH1, TH2... are general theorems about the semantics, they are some of the theorems listed in section 4.3 and 4.5. Theorems named ARITH1, ARITH2... deal with the arithmetic, they are taken from Newey 1973. Theorems named LM1, LM2... are specific lemmas about this program. All of them have been proved in the same environment as the main theorem and their proofs are very simple. Often the proof reduces to a one step simplification. They are:

```
READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,
            FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF))))≡FRAME1(SP,n,x)

ASSUME isnat x ≡ TT, isnat n ≡ TT
```

which implicitly defines the frame FRAME1,

```
MS(body,0,FRAME1(SP,n,x))≡ FRAME1(SP,pred n,x*n)   ASSUME isnat x ≡ TT, ¬(n≡0)≡TT.
```

It specifies the effect of the meaning function MS on the body of the while statement. Moreover

```
MBEXPR(test,0,FRAME1(SP,n,x))≡ ¬(n≡0)   ASSUME isnat n ≡TT, isnat x≡TT
```

evaluates the test appearing in the while, and finally

```
RESULT(WRITE(n1,0,FRAME1(SP,n,x)))≡FACT(n,x)   ASSUME ¬(n≡0)≡FF, isnat(x)≡TT;
```

asserts that, when the loop is over, the value of the varible n1 is FACT(n,x).

As already noted the proof is fearly standard and could be almost completely automated by increasing the proving capabilities of LCF. The case of the goto program the proof is standard as well, but much longer. In fact the theorem presented in 4.3 no longer applies, so the goal to be proved, after the first simplification is:

```
∀n x . isnat(n) :: isnat(x) ::
  RESULT(MS(mkcmpnd(mklabstat(1,mkcond(test,
            mkcmpnd(mkass(n1,mkexpr2(times,n1,n2)),
            mkcmpnd(mkass(n2,mkexpr1(minus1,n2)),
            mkcmpnd(mkgoto(1),ES))),ES)),mkcmpnd(mkwrite(n1),ES)),0,
    READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,
      FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))) ⊂ FACT(n,x).
```

In order to prove it by induction on MS a possibility is that of proving the above goal in parallel with the following 3 goals:

```
∀n x . isnat(n) :: isnat(x) ::
  RESULT([λs.COND(MBEXPR(test,0,s),
          MS(mkcmpnd(mkass(n1,mkexpr2(times,n1,n2)),
            mkcmpnd(mkass(n2,mkexpr1(minus1,n2)),
            mkcmpnd(mkgoto(1),mkcmpnd(mkwrite(n1),ES)))),0,s),
          WRITE(n1,0,s)]
    READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,
      FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))) ⊂ FACT(n,x).

∀n x . isnat(n) :: isnat(x) ::
  RESULT([λs.COND(MBEXPR(test,0,s),
          ASSIGN(n1,MEXPR(mkexpr2(times,n1,n2),0),s)⊗
          MS(mkcmpnd(mkass(n2,mkexpr1(minus1,n2)),
            mkcmpnd(mkgoto(1),mkcmpnd(mkwrite(n1),ES)))),0,s),
          WRITE(n1,0,s)]
    READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,
```

```
        FRAMEØ(FACTORIAL,INPUT(LIST(n,x)),EOF)))))) ⊂ FACT(n,x).

∀n x . isnat(n) :: isnat(x) ::
  RESULT([λs.COND(MBEXPR(test,Ø,s),
            ASSIGN(n1,MEXPR(mkexpr2(times,n1,n2),Ø),s)⊗
            ASSIGN(n2,MEXPR(mkexpr1(minus1,n2)),Ø),s)⊗
            MS( mkcmpnd(mkgoto(1),mkcmpnd(mkwrite(n1),ES))))),Ø,s),
            WRITE(n1,Ø,s)]
     READ(n1,Ø,READ(n2,Ø,CREAV(Ø,n2,INT,Ø,CREAV(Ø,n1,INT,Ø,
       FRAMEØ(FACTORIAL,INPUT(LIST(n,x)),EOF)))))) ⊂ FACT(n,x).
```

In this way there are four induction hypotheses to be instantiated and it can be seen that each of them serves to prove the next goal in the above order. Even this tricky way is standard. It can be applied whenever in a program a backward goto is encountered. In addition, such tactic could also be implemented in a PASCAL oriented version of LCF, so the user is relieved from the task of generating all the parallel goals.

Section 5.2 The McCarthy Airline Reservation System

John McCarthy suggested the problem of proving the correctness of a program for the reservation system of the *McCarthy Airline Company*. Such company has one plane, with only one seat. The plane never flies! There are two customers, each one sometimes makes a reservation and then, tired of waiting for the departure of the plane he cancels. Later on he may try again.

Proving the correctness of a program for the McCarthy Airline reservation system is interesting since it presents some characteristics absent in the programs so far proved correct. A program which realizes a reservation system must deal with a potentially infinite stream of input data "read" at successive instants of time. Each time a request is inputed, an output datum is produced. The correctness of incremental computations cannot be dealt with in a system where the input and output operations aren't mentioned.

Usually, in the existing systems for program verification, I/O is completely ignored. It is not considered to influence the "meaning" of a program. In fact, existing systems deal with algorithms, rather than programs, even though such algorithms are expressed in the syntax of a programming language.

Our axiomatization of PASCAL includes the operations of inputing data from an input file into locations of the store and outputing data from the store into an output file. The length of these files isn't fixed a priori, even for a particular program.

In our formalism we may express and prove a statement of the correctness of a PASCAL program for the McCarthy Airline reservation system. Such statement asserts that, no matter what the sequence of requests has been, the seat at any instant of time is reserved for the right person.

Let
st denote the seat,
wl denote the waiting list,
rq denote the request and
ps denote the passenger.

The variable *st* may assume the values 0, 1 or 2 meaning free, reserved for passenger 1 or 2. The variable *wl* assumes the values 0, 1 and 2 with the same meaning. *rq* may assume the value 0 and 1 for cancellation and reservation, respectively. *ps* assumes the values 1 or 2, denoting the two passengers.

A PASCAL program realizing the McCarthy Airline reservation system is the following:

```
begin
var st,wl,ps,rq: integer;
read(wl);
read(st);
repeat
   begin
   read(rq);
   if rq≠3
    then begin
         read(ps);
         if rq=1
          then if st=0 ∨ st=ps
               then st:=ps else wl:=ps;
          else if st=0 ∨ st≠ps
               then wl:=0 else begin st:=wl end
         write(st)
         end
until rq=3
end
```

The program consists of an initialization part, in which the initial status of the seat and the waiting list (presumably both 0) are inputed, and of a repeat loop. The body of the loop consists in reading new data, updating the status of the seat and the waiting list and then writing the status of the seat into the output buffer. An extraneous value in the input sequence, in this case the number 3, stops the repetition.

This program doesn't make any assumption on the behavior of the passenger or about the kind of requests it receives. Each request is accepted and the program behaves correctly even if, for instance, two cancellations in a row are done by the same person.

The abstract syntax for the above program is:

```
McCARTHY ≡ mktext(DP,SP),

DP ≡ mkcmpnd(mkvardecl(wl,INT),mkcmpnd(mkvardecl(st,INT),
          mkcmpnd(mkvardecl(rq,INT),mkcmpnd(mkvardecl(ps,INT),ES)))),

SP ≡ mkcmpnd(mkread(wl),mkcmpnd(mkread(st),
          mkcmpnd(mkrepeat(BODY,mkrel(eq,rq,mknumconst(3))),ES))),

BODY ≡ mkcmpnd(mkread rq,mkcmpnd(mkcond(mkrel(eq,rq,mknumconst(3)),ES,
                          mkcmpnd(mkread ps,SEATUPDATE)),ES)),

SEATUPDATE≡
```

```
mkcmpnd(mkcond(mkrel(eq,rq,mknumconst 1),
  mkcmpnd(mkcond(mkbexpr2(or,mkrel(eq,st,mknumconst 0),mkrel(eq,st,ps)),
          mkcmpnd(mkass(st,ps),ES),mkcmpnd(mkass(wl,ps),ES)),ES),
    mkcmpnd(mkcond(mkbexpr2(or,mkrel(eq,st,mknumconst 0),mkbexpr1(not,mkrel(eq,st,ps))),
          mkcmpnd(mkass(wl,mknumconst 0,ES),
          mkcmpnd(mkass(st,wl),mkcmpnd(mkass(wl,mknumconst 0,ES))),ES)),
    mkcmpnd(mkwrite st, ES)),
```

The statement of the partial correctness of the McCARTHY program is:

```
∀isq osq p q.iswfsq(isq)::iswfos(osq)::isint(p)::isint(q)::
        APPLY(McCARTHY,p,q,isq,osq)⊂BOOKING(p,q,isq,osq),
```

where: isq denotes the input sequence, osq denotes the initialization of the output buffer, namely the output sequence, p and q are the initial values of the waiting list and the seat.

The predicate iswfsq (is-well-formed-sequence) is defined as:

```
iswfsq ≡ [∝F.[λsq. (el1(sq)= 3)→TT,iseof sq →UU,isrqst(el1 sq)∧isprsn(el2 sq)→F(tail1 sq),FF]],
```

where el1, el2, tail1, isrqst (isrequest) and isprsn (isperson) are defined as follows:

```
el1 ≡ [λx. fstof x],

el2 ≡ [λx. el1(rmdof x)],

tail1 ≡ [λx. rmdof(rmdof x)],

isrqst ≡ [λx.(x=0)∨(x=1)],
isprsn ≡ [λx.(x=1)∨(x=2)].
```

The predicate iswfos (is-well-formed-output-sequence) is:

```
iswfos ≡ [∝F.[λos.iseof os → TT,isint(fstof os)→F(rmdof os),FF]],
```

and must be satisfied by the object, presumably EOF, that initializes the output buffer.

The combinator APPLY appearing in the definition of the goal is:

```
APPLY ≡ [λ p x y is os.FUNCT(p,os,LIST(x,y,is))],

LIST ≡ [λx y is. mkpair(x,mkpair(y,is))],
```

FUNCT, the combinator which "interprets" a program p in the frame where the input and output buffers have been initialized, is described in section 2.

The fact that, at each moment, the seat is reserved for the right person, is expressed in LCF by the function BOOKING:

```
BOOKING = [∝F.[λ st wl sq os.
  iseof sq     → UU,
  (el1 sq=3)   → os,
  F(tail1 sq,stupdt(sq,st,wl),wlupdt(sq,st,wl),mkpair(stupdt(sq,st,wl),os))]],
```

where stupdt (seatupdate) and wlupdt (waiting-listupdate) are defined as:

```
stupdt=[λsq st wl.(el1 sq=1)→(st=0)v(st=el2 sq)→el2 sq,st,(st=0)v ¬(st=el2 sq) → st,wl],

wlupdt=[λsq st wl.(el1 sq=1)→(st=0)v(st=el2 sq)→wl,el2 sq,0].
```

We express the fact that, at each instant of time the program "answers" in the right way, by stating that it behaves correctly on input sequences of any length. Being extensional our semantics cannot express the concept of elapsation of time, but, by talking of sequences of any length we give an adequate extensional representation of a continuing process.

The list of LCF commands and the printout of the proof of the partial correctness of the McCARTHY program with respect to the BOOKING function is given in appendix 8. The goal to be proved, after the first simplification is:

```
∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) ::
OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))))=3)→
   REPEAT(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3))),0),0,
   MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),
   MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))) ⊂ BOOKING(p,q,isq,osq)
```

In achieving this goal the theorem on the repeat statement, given in section 4.3 has been used. The combinator FRAME1 is introduced to increase the readability of the goal. It describes the store after the declarations are done.

```
FRAME1 = [λx y sq os. [λf.(f=0)→[λloc.
            loc=typeloc ps→INT,
            loc=typeloc rq→INT,
            loc=typeloc st→INT,
            loc=typeloc wl→INT,
            loc=fileloc INP→INTERNALREP(LIST(x,y,sq)),
            loc=fileloc OUT→INTERNALREP os,
            loc=textloc    →SP,UNDEF],UU].
```

The proof of the McCARTHY program differs from that of the factorial program mainly for two reasons: 1) the while and the repeat statements behave differently, having the test performed at different places. 2) here an initialization is done within the body of the repetition statement. In fact, the two values of rq and ps are read within the loop. For this reason the loop must be executed once in order to create a location named rq and one named ps, before doing an induction on the combinator REPEAT. The goal is proved by cases on the test which controls the repeat loop. The only nontrivial case is that in which the input sequence is not yet over, namely rq≠3. In this case the repeat loop goes on, so an induction is needed for completing the proof. The base case of this induction is trivial. The induction step is proved by doing again cases on the test which establishes the exit conditions from the loop. If the loop is completed a lemma is used to state the result, if it goes on the goal is proved by an appropriate instantiation of the induction hypothesis.

As in the proof of the factorial program the theorems used in the proof have been divided into THs, ARITHs and LMs. THs state facts about the semantics, one of them is the above mentioned theorem about the semantics of the repeat statement for goto-free programs. They are shown in 4.3 and 4.5. ARITHs are theorems dealing with the arithmetic and properties derived from the above axioms on the well formedness of input and output sequences. LMs are specific lemmas regarding this program. The list of these lemmas follows.

```
∀sq os x1 x2. MD(DP,Ø,FRAMEØ(McCARTHY,INPUT(LIST(x1,x2,sq)),INTFRNALREP(os)))≡
                              FRAME1(x1,x2,sq,os);
```

is an implicit definition of FRAME1. It defines the store after the declarations are done.

```
READ(st,Ø,READ(wl,Ø,FRAME1(x1,x2,sq,os)))≡FRAME2(x1,x2,sq,os)

ASSUME iswfsq(sq)≡TT, iswfos(os)≡TT, isint(x1)≡TT, isint(x2)≡TT
```

This statement is an implicit definition of FRAME2. It describes the store after wl and st are initialized.

```
FRAME2 ≡ [λx1 x2  sq os. [λf.(f≡Ø)→[λloc.
            loc≡st        →x2,
            loc≡wl        →x1,
            loc≡typeloc ps→INT,
            loc≡typeloc rq→INT,
            loc≡typeloc st→INT,
            loc≡typeloc wl→INT,
            loc≡fileloc INP→INTERNALREP(sq),
            loc≡fileloc OUT→INTERNALREP(os),
            loc≡textloc    →SP,UNDEF],UU],
```

The next theorem:

```
OUTPUT(MS(BODY,Ø,FRAME2(x1,x2,sq,os)))≡BOOKING(x1,x2,sq,os)

ASSUME ¬(el1 sq = 3 )≡FF,iswfsq sq≡TT,iswfos os≡TT,isint x1≡TT,isint x2≡TT
```

states that, when the input sequence is over, the content of the output file after the execution of BODY in the store described by FRAME2, equals the value of the function BOOKING.

```
BOOKING(stupdt(sq,x,y),wlupdt(sq,x,y),tail1 sq,mkpair(stupdt(sq,x,y),os))≡BOOKING(x,y,sq,os)

ASSUME iswfsq sq ≡TT,iswfos os ≡ TT,isint x ≡ TT, isint y ≡ TT,¬(el1 sq=3)≡TT
```

states a simple property of the function BOOKING.

```
MS(BODY,Ø,FRAME2(stupdt(sq,x,y),wlupdt(sq,x,y),tail1 sq,mkpair(stupdt(sq,x,y),os)))≡
            MS(BODY,Ø,FRAME3(x,y,sq,os))

ASSUME iswfsq sq ≡TT,iswfos os ≡ TT,isint x ≡ TT, isint y ≡ TT,¬(el1 sq=3)≡ TT;

MS(BODY,Ø,FRAME2(x,y,sq,os))≡ FRAME3(x,y,sq,os)
```

```
ASSUME iswfsq sq =TT,iswfos os = TT,isint x = TT, isint y = TT,¬(el1 sq=3)= TT;
```

The two above theorems use the combinator FRAME3 to describe an intermediate store

```
FRAME3 = [λx1 x2 sq os. [λf.(f=ϑ)→[λloc.
          loc=ps        →el2 sq,
          loc=rq        →el1 sq,
          loc=st        →stupdt(sq,x1,x2),
          loc=wl        →wlupdt(sq,x1,x2),
          loc=typeloc ps→INT,
          loc=typeloc rq→INT,
          loc=typeloc st→INT,
          loc=typeloc wl→INT,
          loc=fileloc INP→tail1(INTERNALREP sq),
          loc=fileloc OUT→mkpair(mknumconst stupdt(sq,x1,x2),INTERNALREP os),
          loc=textloc   →SP,UNDEF],UU];
```

FRAME3 is the description of the store after the body of the loop has been executed once.

```
MEXPR(rq,ϑ,MS(BODY,ϑ,FRAME3(x,y,sq,os)))= el3 sq

ASSUME iswfsq sq =TT,iswfos os = TT,isint x = TT,isint y = TT,¬(el1 sq=3) = TT

MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3))),ϑ,MS(BODY,ϑ,FRAME3(x,y,sq,os)))= ¬(el3 sq = 3)

ASSUME iswfsq sq =TT,iswfos os = TT,isint x = TT,isint y = TT,¬(el1 sq=3)= TT

MEXPR(rq,ϑ,MS(BODY,ϑ,FRAME2(x,y,sq,os)))=el1 sq

ASSUME iswfsq sq=TT,iswfos os =TT,isint x=TT,isint y=TT.
```

The three above lemmas are introduced to abbreviate the evaluation of expressions.

SECTION 6 CONCLUSION

The most important aspect of this memo relates to our attempt to axiomatize *all* of the arithmetic part of PASCAL. This is interesting for two reasons. First we are able to describe in LCF different programming language features and show how they interact. Secondly we can express property of classes of programs and use them as lemmas in proofs of theorems about particular programs. A typical example is the theorem about goto-free programs in section 4.2. It is used in section 5.2 to simplify the first proof of the correctness of the factorial program. When interpreted literally, it proves that for goto-free programs the composition rule in Hoare 1969 is valid. By formulating the validity of this rule as a theorem we can discuss, in LCF, the relative merits of various programming features. This has not previously been accessible to a formal treatment, and is important if the mathematical theory of computation is ever to have an effect on language design.

Our desire to axiomatize all aspects of a programming language is not simply a matter of choice of available formalisms but represents a philosophy about what kinds of questions the mathematical theory of computation should ask. The method of attaching inductive assertions to programs treats programs one at a time. We do not think general theories about programs can be developed in this way. Of course using inductive assertions is not a waste of time, but formalisms which use them should be expanded to include more general applicability.

The kind of questions about programs we have in mind include: will it run at all, even if its algorithm is correct? Will it compile? Does some other coding or "optimization" compute the same function? We believe that LCF is capable of expressing these notions. Furthermore, any formalism for describing a programming language could reasonably be expected to have this property.

We criticize the original description of PASCAL, not because Wirth didn't have philosophically reasonable ideas of what various features of a programming language should do, but rather he lacked a formalism which was strong enough to describe the effect of putting together features, which although separately make clear sense, cause problems when combined. The example of the procedure in the discussion of the for statement is a case in point. It is *not* a PASCAL procedure as the value of the index variable of the for statement is changed in its body. This fact, however is hard to detect and is certain to be missed by most compilers. The difficulty arises out of the desire not to make the index of a for statement local to that statement, to have the limits of the for loop variable determined once and for all and to have recursive procedures in the same language. Features when combined in arbitrary ways make even the recognition of well formed programs complicated. Further evidence of this difficulty is found in the large number of restrictions Igarashi, London and Luckham 1973 have put on the application of their rules. The only example of a procedure given in Hoare and Wirth 1973 cannot be treated in their system. It does not seem obvious to us how to extend their style of axiomatization to all of PASCAL. We do not impose any of their restrictions, but describe the full generality allowed by Wirth. The expressive power of LCF permits us to represent their restrictions and to prove that rules similar to theirs are valid for the subset of PASCAL they treat.

The above should reflect on language design. One overwhelming feeling of all three authors after doing this work was that we know large amounts more about how to describe a language to make proving theorems about it reasonable. We believe that the ability to describe programming features and demonstrate by proving theorems that a language has certain properties represents a particularly satisfying way to describe a language. Furthermore we propose this as a standard for acceptable descriptions.

One possible idea for future work is designing a programming language using the more precise description of this paper. Only small modifications to PASCAL are necessary to give a similar language a demonstrably smoother semantics. Thus, by starting with a more detailed description, some properties of the language, which could only be informally described before would now be made explicit as statements in LCF. One could then begin to amass a collection of theorems that could be used to prove properties of particular programs. We could then integrate everything into an LCF-PASCAL "machine" which took a concrete PASCAL syntax and generated the LCF abstract syntactic representation. Of course the new language would have to include more features than those discussed here. Obvious candidates are real arithmetic, file manipulation and more complicated data structures. If we wanted to abandon the ALGOL like control structures it would be possible to choose either that of LISP or even the more aggressive control structures of Bobrow and Wegbreit or the Landin J operator. It would be an interesting project to describe them all and see what theorems hold when you allow them to exist simultaneously.

We chose to work out the McCarthy airline reservation system as an example because we believe the treatment of interactive programs is another area which a vital mathematical theory of computation must consider. Our idea for how to treat the correctness of continously interactive programs was to consider them as functions from sequences of inputs to sequences of outputs. If the processes you are considering are continous, that is, some initial sequence of outputs is completely determined after some fixed number of inputs, then equivalently we can consider the correctness of finite output sequences given finite input sequences. Basically this idea has been used in intuitionistic theories of free choice sequence as developed by Brouwer and Kleene (see Kleene and Vesley 1965).

We end this memo with some comments about LCF. A major difficulty involved in using LCF as the language for interpreting programming languages is that descriptions of the data being manipulated (in our case integers) is awkward. The axiomatization of arithmetic in LCF although adequate is both non standard and frequently hard to use. It is partially the fault of LCF as it does not implement such nice user oriented features as arbitrary structural inductions. It forces you to use computation induction in its primitive form. Unfortunately the implementation cannot be blamed for everything. A proof of Wilson's theorem, for example, would be virtually impossible even by mathematical induction. LCF terms not only have interpretations as functions, but can also be interpreted as computation rules. Although this duality has not been fully exploited it is the essential reason that the simplification mechanism of LCF is so successful.

APPENDIX 1

A BRIEF DESCRIPTION OF LCF

The syntax of LCF sentences is described in detail in Milner 1972a. Here we only give an informal description of the language, its interpretation and enough of the abbreviation conventions to make the formulas in this report intelligible to those not familiar with LCF.

There are two kinds of base variables and constants in LCF. Those that range over individuals and those that range over truth values. Each term has an associated type. If t is a term and σ its associated type symbol we write $t:\sigma$. IND and TV are type symbols. If σ and τ are type symbols then so is $(\sigma\rightarrow\tau)$. We write x:IND and x:TV for x of type individual and truth values respectively. There are variables and constants for each different type symbol. The variable symbols of different types are supposed to be disjoint. There are three constants of type TV. They are TT for true, FF for false, and UU for undefined.

Terms are formed as follows: if $x:\sigma$ is a variable and $t:\tau$ then $[\lambda x.t]:(\sigma\rightarrow\tau)$ is a term whose interpretation is a function from things of type σ onto things of type τ. In LCF $[\lambda x.[\lambda y.t]]$ is abbreviated by $[\lambda x\ y.t]$. If $r:(\sigma\rightarrow\tau)$ and $s:\sigma$ then $r(s):\tau$. We interpret $r(s)$ as the result of applying the function r to the argument s. We frequently write this $r\ s$, thus

$$a\ b\ c \equiv a(b)(c) \equiv (a(b))(c) \equiv a(b,c).$$

Note that if τ is TV then r is a predicate. Conditional expressions are formed as $(p\rightarrow q,r)$, where p:TV and q, r are of the same type. On the undefined truth value the conditional is undefined, i.e. for all q and r, $(UU\rightarrow q,r)\equiv UU$. Terms are also built up using the least fixed point operator $\propto$. If $x:\sigma$ is a variable and $s:\sigma\rightarrow\sigma$ then $[\propto x.s]$ is a term representing the least fixed point of the functional s.

Atomic well formed formulas (or AWFFs) are formed by joining two terms using $\equiv$ or $\subset$, i.e. if r and s are terms then $r\equiv s$ and $r\subset s$ are AWFFs. $r\equiv s$ means that the functions denoted by r and s are the same. In a full description of the theory there is also a partial order between terms of the same type. This is represented using $\subset$.

The more usual definition of the factorial function *fact(n) ← if x=0 then 1 else n*fact(n-1)* becomes in LCF

$$FACT \equiv [\propto f.[\lambda n.(n\equiv 0\rightarrow 1, n*f(n-1))]].$$

LCF also allows two other abbreviations.

$\forall x.f\equiv g$ is the same as $[\lambda x.f]\equiv[\lambda x.g]$.

Because terms are interpreted as extensionally given functions, this definition makes sense.

$P::Q\equiv R$ is the same as $(P\rightarrow Q,UU)\equiv(P\rightarrow R,UU)$

Intuitively this is read as: If P is true then $Q\equiv R$, otherwise I don't know anything.

APPENDIX 2

THE ABSTRACT SYNTAX

2.1 Syntax for Statements

AXIOM SYNAXS:

```
∀ d s. type(mktext d s) = _T,
∀ d s. declof(mktext d s) = d,
∀ d s. statmof(mktext d s) = s,

∀ d1 d2. type(mkcmpnd d1 d2) = _CM,
∀ d1 d2. fstof(mkcmpnd d1 d2) = d1,
∀ d1 d2. rmdof(mkcmpnd d1 d2) = d2,

∀ n ty. type(mktypedef n ty) = _TD,
∀ n ty. namof(mktypedef n ty) = n,
∀ n ty. typof(mktypedef n ty) = ty,

∀ n1 n2. type(mksublim n1 n2) = _SL,
∀ n1 n2. lbof(mksublim n1 n2) = n1,
∀ n1 n2. ubof(mksublim n1 n2) = n2,

∀ ai ty. type(mkarspec ai ty) = _AS,
∀ ai ty. arlimof(mkarspec ai ty) = ai,
∀ ai ty. typeiof(mkarspec ai ty) = ty,

∀ i1 i2. type(mkpair i1 i2)= _PA,
∀ i1 i2. fstof(mkpair i1 i2)= i1,
∀ i1 i2. rmdof(mkpair i1 i2)= i2,

∀ n ty. type(mkvardecl n ty) = _VD,
∀ n ty. namof(mkvardecl n ty) = n,
∀ n ty. typof(mkvardecl n ty) = ty,

∀ n ps. type(mkprocdecl n ps) = _PD,
∀ n ps. namof(mkprocdecl n  ps) = n,
∀ n ps. prspof(mkprocdecl n ps) = ps,

∀ n fs ty. type(mkfundecl n fs ty) = _FD,
∀ n fs ty. namof(mkfundecl n  fs ty) = n,
∀ n fs ty. fnspof(mkfundecl n fs ty) = fs,
∀ n fs ty. typeof(mkfundecl n fs ty) = ty,

∀ f t.type(mkprocspec f t) = _PS,
∀ f t.fargof(mkprocspec f t) = f,
∀ f t.textof(mkprocspec f t) = t,

∀ f t.type(mkfunspec f t) = _FS,
∀ f t.fargof(mkfunspec f t) = f,
∀ f t.textof(mkfunspec f t) = t,
```

```
∀ x ty. type(mkvarp x ty) ≡ _VRP,
∀ x ty. namof(mkvarp x ty) ≡ x,
∀ x ty. typof(mkvarp x ty) ≡ ty,

∀ x ty. type(mkvalp x ty) ≡ _VLP,
∀ x ty. namof(mkvalp x ty) ≡ x,
∀ x ty. typof(mkvalp x ty) ≡ ty,

∀ x ty. type(mkfunp x ty) ≡ _FP,
∀ x ty. namof(mkfunp x ty) ≡ x,
∀ x ty. typof(mkfunp x ty) ≡ ty,

∀ x. type(mkprocp x) ≡ _PP,
∀ x. namof(mkprocp x) ≡ x,


∀ l s. type(mklabstat l s) ≡ _LS,
∀ l s. labelof(mklabstat l s) ≡ l,
∀ l s. statmof(mklabstat l s) ≡ s,

∀ n. type(mkread n) ≡ _RD,
∀ n. namof(mkread n) ≡ n,

∀ n. type(mkwrite n) ≡ _WT,
∀ n. namof(mkwrite n) ≡ n,

∀n. type(mkgoto n) ≡ _G,
∀n. labelof(mkgoto n) ≡ n,

∀n e. type(mkass n e) ≡ _A,
∀n e. lhsof(mkass n e) ≡ n ,
∀n e. rhsof(mkass n e) ≡ e ,

∀ n a. type(mkproccall n a ) ≡ _PC,
∀ n a. namof(mkproccall n a ) ≡ n,
∀ n a. actargof(mkproccall n a) ≡ a,

∀be p1 p2. type(mkcond be p1 p2) ≡ _C,
∀be p1 p2. testof(mkcond be p1 p2) ≡ be,
     p1 p2. thenof(mkcond be p1 p2) ≡ p1,
    p1 p2. elsoof(mkcond e p1 p2) ≡ p2,

∀t b. type(mkwhile t b) ≡ _W,
∀t b. testof(mkwhile t b) ≡ t ,
∀t b. bodyof(mkwhile t b) ≡ b,

∀b t. type(mkrepeat b t) ≡ _R,
∀b t. bodyof(mkrepeat b t) ≡ b,
∀b t. testof(mkrepoat b t) ≡ t ,

∀i e1 e2 b. type(mkforto i e1 e2 b)≡_FT,
∀i e1 e2 b. indexof(mkforto i e1 e2 b)≡ i,
∀i e1 e2 b. lbof(mkforto i e1 e2 b)≡ e1,
∀i e1 e2 b. ubof(mkforto i e1 e2 b)≡ e2,
∀i e1 e2 b. bodyof(mkforto i e1 e2 b)≡ b,
```

```
∀i e1 e2 b. type(mkfordn i e1 e2 b)≡_FD,
∀i e1 e2 b. indexof(mkfordn i e1 e2 b)≡ i,
∀i e1 e2 b. ubof(mkfordn i e1 e2 b)≡ e1,
∀i e1 e2 b. lbof(mkfordn i e1 e2 b)≡ e2,
∀i e1 e2 b. bodyof(mkfordn i e1 e2 b)≡ b,


  type UU ≡ UU,
  type ES ≡ _ES,
  type EOF ≡ _EOF;
```

2.2 Syntax for Expressions

AXIOM EXPRAX:

```
  ∀o e1. type(mkexpr1 o e1) ≡ _E,
  ∀o e1. opof(mkexpr1 o e1) ≡ o,
  ∀o e1. arg1of(mkexpr1 o e1) ≡ e1,

  ∀bo be1. type(mkbexpr1 bo be1) ≡ _BE,
  ∀bo be1. bopof(mkbexpr1 bo be1) ≡ bo,
  ∀bo be1. barg1of(mkbexpr1 bo be1) ≡ be1,

  ∀o e1 e2. type(mkexpr2 o e1 e2) ≡ _E,
  ∀o e1 e2. opof(mkexpr2 o e1 e2) ≡ o,
  ∀o e1 e2. arg1of(mkexpr2 o e1 e2) ≡ e1,
  ∀o e1 e2. arg2of(mkexpr2 o e1 e2) ≡ e2,

  ∀bo be1 be2. type(mkbexpr2 bo be1 be2) ≡ _BE,
  ∀bo be1 be2. bopof(mkbexpr2 [illegible] be2) ≡ bo,
  ∀bo be1 be2. barg1of(mkbexp[illegible] be2) ≡ be1,
  ∀bo be1 be2. barg2of(mkbexpr[illegible] be1 be2) ≡ be2,

  ∀bo e1 e2. type(mkrel bo e1 e2) ≡ _BE,
  ∀bo e1 e2. bopof(mkrel bo e1 e2) ≡ bo,
  ∀bo e1 e2. arg1of(mkrel bo e1 e2) ≡ e1,
  ∀bo e1 e2. arg2of(mkrel bo e1 e2) ≡ e2,

  ∀ n i. type(mkae n i) ≡ _AE,
  ∀ n i. namof(mkae n i) ≡ n,
  ∀ n i. subof(mkae n i) ≡ i,

  ∀ n a. type(mkfundes n a) ≡ _FA,
  ∀ n a. namof(mkfundes n a) ≡ n,
  ∀ n a. actargof(mkfundes n a) ≡ a,

  ∀ n. type(mknumconst n) ≡ _NC,
  ∀ n. numof(mknumconst n) ≡ n;
```

2.3 Predicates for the Identification of Syntactic Constructs

AXIOM PREDAX:

```
∀x.. istext x ≡ type x = _T,
∀x. iscmpnd x ≡ type x = _CM,
∀x. istypedef x ≡ type x = _TD,
∀x. issublim x ≡ type x = _SL,
∀x. isarspec x ≡ type x = _AS,
∀x. ispair x ≡ type x = _PA,
∀x. isvardecl x ≡ type x = _VD,
∀x. isprocdecl x ≡ type x = _PD,
∀x. isfundecl x ≡ type x = _FD,
∀x. isprocspec x ≡ type x = _PS,
∀x. isfunspec x ≡ type x = _FS,
∀x. isvarp x ≡ type x = _VRP,
∀x. isvalp x ≡ type x = _VLP,
∀x. isfunp x ≡ type x = _FP,
∀x. isprocp x ≡ type x = _PP,

∀x. islabstat x ≡ type x = _LS,
∀x. isread x ≡ type x = _RD,
∀x. iswrite x ≡ type x = _WT,
∀x. isgoto x ≡ type x = _G,
∀x. isass x ≡ type x = _A,
∀x. isproccall x ≡ type x = _PC,
∀x. iscond x ≡ type x = _C,
∀x. iswhile x ≡ type x = _W,
∀x. isrepeat x ≡ type x = _R,
∀x. isforto x ≡ type x = _FT,
∀x. isfordn x ≡ type x = _FD,

∀x. isemptyst x ≡ type x = _ES ,
∀x. iseof x ≡ type x = _EOF,

∀x. isconst x ≡ type x = _NC,
∀x. isname x ≡ type x = _N,
∀x. isexpr x ≡ type x = _E,
∀x. isbexpr x ≡ type x = _BE,
∀x. isrel x ≡ type x = _BE,
∀x. isae x ≡ type x = _AE,
∀x. isfundes x ≡ type x = _FA;
```

2.4 Auxiliary Predicates and Functions

```
AXIOM AUXSYN :

isname FUNV ≡ FF,

fstof EOF ≡ UU,

rmdof EOF ≡ UU;

issingle ≡ [λst. (isread st)∨(iswrite st)∨(issimple st)∨(isemptyst st)],

issimple ≡ [λst. (isgoto st)∨(isass st)∨(isproccall st)],

fortest ≡ [λx .isforto(x)→mkrel(lseq,lbof(x),ubof(x)),isfordn(x)→mkrel(greq,ubof(x),lbof(x)),UU] ,

fortoup ≡ [λx .mkcmpnd(mkforto(indexof(fstof(x)),mkexpr1(plus1,indexof(fstof(x))),
                    ubof(fstof(x)),bodyof(fstof(x))),rmdof(x))],

fordnup ≡ [λx .mkcmpnd(mkfordn(indexof(fstof(x)),mkexpr1(minus1,indexof(fstof(x))),
                    lbof(fstof(x)),bodyof(fstof(x))),rmdof(x))],

isrepwh  ≡ [λst. (isrepeat st)∨(iswhile st)],

isiter   ≡ [λst. (isforto st)∨(isfordn st)∨(isrepwh st)],

isparameter ≡ [λx. (isvarp x)∨(isvalp x)∨(isprocp x)∨(isfunp x)],

isbasetype ≡ [λn (n=INT)∨(type(n)=_SL)],

istyppart ≡ [λn.ispair(n)∨iseof(n)],

occurs  ≡ [∝F.[λn st.
          isemptyst st   → UU,
          iscmpnd st     → F(n,fstof st)∨F(n,rmdof st),
          islabstat st   → (n=labelof st)→TT,F(n,rmdof st),
          issingle st    → FF,
          isiter st      → F(n,bodyof st),
          iscond st      → F(n,thenof st)∨F(n,elseof st),UU]],

append ≡ [∝F.[λ st1 st2.
          isemptyst st1 → st2,
          iscmpnd st1 → mkcmpnd(fstof st1, F(rmdof st1,st2)),UU]],

segm ≡ [∝F.[λn st.
       isemptyst st → UU,
       iscmpnd st→
        isemptyst st      →F(n,rmdof st),
        islabstat(fstof st)→(n=labelof st)→ st,F(n,mkcmpnd(statmof(fstof st),rmdof st)),
        issingle(fstof st) →F(n,rmdof st),
        iscond(fstof st) →occurs(n,thenof(fstof st))→append(F(n,thenof(fstof st)),rmdof st),
                          occurs(n,elseof(fstof st))→append(F(n,elseof(fstof st)),rmdof st),
                          F(n,rmdof st),
        isrepwh(fstof st)→occurs(n,bodyof(fstof st))→append(F(n,bodyof(fstof st)),st),
                          F(n,rmdof st),
        isforto(fstof st)→occurs(n,bodyof(fstof st))→append(F(n,bodyof(fstof st)),fortoup(st)),
```

```
                    F(n,rmdof st),
        isfordn(fstof st)→occurs(n,bodyof(fstof st))→append(F(n,bodyof(fstof st)),fordnup(st)),
                    F(n,rmdof st),UU,UU]],

isvariable ≡ [λx.isname(x)∨isae(x)],

isunary ≡ [λx.(x=pplus)∨(x=pminus)∨(x=plus1)∨(x=minus1)],

isbunary ≡ [λx.(x=not)],

isbinary ≡ [λx.(x=plus)∨(x=minus)∨(x=times)∨(x=div)∨(x=rmdr)∨(x=and)∨(x=or)∨
            (x=lseq)∨(x=greq)∨(x=lt)∨(x=gt)∨(x=eq)∨(x=neq)],

isbbinary ≡ [λx.(x=and)∨(x=or)],

isrelop ≡ [λx.(x=lseq)∨(x=greq)∨(x=lt)∨(x=gt)∨(x=eq)∨(x=neq)];
```

APPENDIX 3

THE SEMANTICS

3.1 Top Level Functions

```
AXIOM TOPSEM:

FUNCT    ≡ [λp o i.(INPUT⊗PASCAL(p,o)⊗OUTPUT)(i)],

PASCAL   ≡ [λp o i. MP(p,Ø,FRAMEØ(p,o,i))],

FRAMEØ   ≡ [λt i o f. (f=Ø)→[λloc.(loc=filoloc INP) → INTERNALREP(i),
                              (loc=filoloc OUT) → INTERNALREP(o),
                              (loc=textloc) → statmof t,UNDEF],UU],

MP       ≡ [λt f. MD(declof t,f)⊗MS(statmof t,f)],

INPUT    ≡ ID,

OUTPUT   ≡ [∝F.[λs.[λi.iseof  i →EOF,
                    ispair i →mkpair(F(fstof i),F(rmdof i)),
                    isconst i→numof(i),UU](OBUFFER s)]],

INTERNALREP ≡ [∝F.[λi.iseof  i →EOF,
                   ispair i →mkpair(F(fstof i),F(rmdof i)),
                   isint  i →mknumconst(i),UU]];
```

3.2 Declaration Part

```
AXIOM DECSEM:

MD      = [λd f. MDEF(d,f)⊗MDEC(d,f)],

MDEF    = [∝F [λd f.isemptyst  d → ID,
                istypedef  d → CREAT(f,namof d,typof d),
                iscmpnd    d → F(fstof d,f)⊗F(rmdof d,f),ID]],

MDEC    = [∝F.[λd f.isemptyst  d → ID,
                isvardecl  d → CREAV(f,namof d,typof d,f),
                isprocdecl d → CREAP(f,namof d,prspof d,f),
                isfundecl  d → CREAF(f,namof d,fnspof d,typeof d,f,f),
                iscmpnd    d → F(fstof d,f)⊗F(rmdof d,f),ID]],

CREAT   = [λf n ty s.CREALOC(f,s,typidloc,n,ty)],

CREAV   = [λf n ty fl s.CREALOC(f,s,typeloc,n,TYPEVAL(ty,fl,s))],

CREAP   = [λf n ps fl s.STORE(f,CREALOC(f,s,acclnk,n,fl),procloc n,ps)],

CREAF   = [λf n fs ty ft fl s.
          STORE(f,STORE(f,CREALOC(f,s,acclnk,n,fl),typfunloc n,TYPEVAL(ty,ft,s)),funcloc n,fs)],

CREALOC = [λf s loc n val.ISPRESENT(n,s(f))→UU,STORE(f,s,loc n,val)];
```

3.3 Definition of MS

AXIOM MSDEF:

```
MS=[μF.[λst f.
   isemptyst st    → ID,
   iscmpnd st      →
    isemptyst(fstof st)→ F(rmdof st,f),
    islabstat(fstof st)→ F(mkcmpnd(statmof(fstof st),rmdof st),f),
    isgoto(fstof st)   → GOTO(F,labelof(fstof st),f),
    isass (fstof st)   → ASSIGN(lhsof(fstof st),MEXPR(rhsof(fstof st),f),f)⊗F(rmdof st,f),
    isproccall(fstof st)→[λs.MPB(PROCFAL(namof(fstof st),f,s),actargof(fstof st),f,s,namof(fstof st))]⊗
                         [λs.MD(PROCDECL(namof(fstof st),f,s),succ f,s)]⊗
                           [λs.F(PROCBODY(namof(fstof st),f,s),succ f,s)]⊗CLEAR(succ f)⊗F(rmdof st,f),
    isread(fstof st)   → READ(namof(fstof st),f)⊗F(rmdof st,f),
    iswrite(fstof st)  → WRITE(namof(fstof st),f)⊗F(rmdof st,f),
    iscond(fstof st)   → COND(MBEXPR(testof(fstof st),f),
                         F(append(thenof(fstof st),rmdof st),f),F(append(elseof(fstof st),rmdof st),f)),
    iswhile(fstof st)  → COND(MBEXPR(testof(fstof st),f),
                         F(append(bodyof(fstof st),st),f),F(rmdof st,f)),
    isrepeat(fstof st) → F(append(bodyof(fstof st),mkcmpnd(mkcond(mkbexpr1(not,
                              testof(fstof st)),fstof st,ES),rmdof st)),f),
    isforto(fstof st)  → COND(MBEXPR(fortest(fstof st),f),
                         ASSIGN(indexof(fstof st),MEXPR(lbof(fstof st),f),f)⊗
                         F(append(bodyof(fstof st),fortoup st),f),F(rmdof st,f)),
    isfordn(fstof st)  → COND(MBEXPR(fortest(fstof st),f),
                         ASSIGN(indexof(fstof st),MEXPR(ubof(fstof st),f),f)⊗
                         F(append(bodyof(fstof st),fordnup st),f),F(rmdof st,f)),    UU,UU]];
```

3.4 Axioms for Statements

```
AXIOM STATSEM:

READ   ≡ [λn f s.ISFUNFR(f,s,∅)→ASSIGN(n,MEXPR(fstof(IBUFFER s),f),f,
                          STORE(0,s,fileloc INP,rmdof(IBUFFER s))),UU],

WRITE  ≡ [λn f s.ISFUNFR(f,s,∅)→ STORE(∅,s,fileloc OUT,
                    mkpair(mknumconst(FFTCHV(n,f,s)),OBUFFER s)),UU],

GOTO   ≡ [λF.[λn f. F(segm(n,TEXT(f)),f)]],

ASSIGN ≡ [∝F.[λn v f s.
           n=FUNV→ISADMISVAL(s(f,typeloc FUNV),v(s))→STORE(f,s,FUNV,v(●)),UU,
           ISINTYPE(n,v,f,s)→STORE(f,s,LOCOFVAR(n,f,s),v(s)),
           istopf(f)→UU,
           ISFUNFR(f,s,NEWFP(n,f,s))→F(VARBNDTO(n,f,s),v,NEWFP(n,f,s),s),UU]],

COND   ≡ [λq f g s.(q(s)→f(s),g(s))],

MPB    ≡ [λfa aa f s n.BIND(fa,aa,succ f,
            MAKFRAME(PROCBODY(n,f,s),PFLNK(n,f,s),succ f,s))],

CLEAR  ≡ [λf s f1.(f1=f)→UU,s(f1)];
```

3.5 Binding Mechanism

AXIOM BINDINGS:

```
BIND  = [∝F.[λfa aa f s.
        isecf fa → (iseof aa → s,UU),
        isparameter(fstof fa)→F(rmdof fa,rmdof aa,f,MKBINDING(fstof fa,fstof aa,f,s)),UU]],

MKBINDING = [λfa aa f s.
        isvarp(fa) → TYMATCH(fa,typeloc,aa,f,s) →CREALOC(f,s,bindloc,namof fa,EXPRFORV(aa)),UU,
        isvalp(fa) → ASSIGN(namof fa,MEXPR(aa,f),f,CREAV(f,namof fa,typof fa,CRNTF(f,s),s)),
        isfunp(fa) → TYMATCH(fa,typfunloc,aa,f,s) →
                CREAF(f,namof fa,FUNCDEF(aa,f,s),typof fa,CRNTF(f,s),PFLINK(aa,f,s),s),UU,
        isprocp(fa)→ CREAP(f,namof fa,PROCDEF(aa,f,s),PFLINK(aa,f,s),s),UU],

TYMATCH = [λfa loc aa f s.TYPEVAL(typof fa,CRNTF(f,s),s)=TYPEDEF(loc aa,pred f,s)],

TYPEVAL = [∝F.[λn f s.
        isbasetype n → n,
        isarspec  n → mkarspec(F(arlimof n,f,s),F(typeIof n,f,s)),
        istyppart n → iseof n  → n,ispair n → mkpair(F(fstof n,f,s),F(rmdof n,f,s)),UU,
        ISLOCAL(typidloc n,s(f))→F(s(f,typidloc n),f,s),
        istopf f → UU,F(n,CRNTF(f,s),s)]];
```

3.6 Evaluation of Expressions

AXIOM EXPRESSIONS:

```
MEXPR ≡ [∝F.[λe f s.
        isconst e     → MCONST e,
        isvariable e → FETCHV(e,f,s),
        isfundes e    → RETURN(succ f,MF(namof e,actargof e,f,s)),
        isexpr e →isunary(opof e) → MOP1(opof e,F(arg1of e,f,s)),
                  isbinary(opof e)→ MOP2(opof e,F(arg1of e,f,s),F(arg2of e,f,s)),UU,UU]],

MF    ≡ [λn a f. MFB(FUNCFAL(n,f),a,f,n)⊛MP(FUNCDEF(n,f),succ f)],

MFB   ≡ [λfa aa f n s.BIND(fa,aa,succ f,CREALOC(succ f,typeloc,FUNV,TYPEDEF(n,f,s),
                          MAKFRAME(FUNCBODY(n,f,s),PFLNK(n,f,s),succ f,s) ))],

MBEXPR ≡ [∝F.[λe f s.
     (e=true)→TT,(e=false)→FF,
     isbexpr e →isbunary(bopof e) → MBOP1(bopof e,F(barg1of e,f,s)),
                isbbinary(bopof e)→ MBOP2(bopof e,F(barg1of e,f,s),F(barg2of e,f,s)),
                isrelop(bopof e)→RELOP(bopof e,MEXPR(arg1of e,f,s),MEXPR(arg2of e,f,s)),UU,UU]],

MCONST ≡ [λx.isconst x → numof x,UU],
MOP1   ≡ [λx.x=pplus→λx.x,x=pminus→λx.(0-x),x=plus1→succ,x=minus1→pred,UU],
MBOP1  ≡ [λx.x=not→¬,UU],

MOP2   ≡ [λx.x=plus→!+,x=minus→!-,x=times→!*,x=div→!/,x=rmdr→mod,UU],
MBOP2  ≡ [λx.x=and→!∧,x=or→!∨,UU],
RELOP  ≡ [λx.x=leeq→!≤,x=greq→!≥,x=lt→!<,x=gt→!>,x=eq→!=,x=neq→≠,UU];
```

3.7 Variables

AXIOM VARIABLES:

```
NAMOFVAR = [λv.n=FUNV→UU,isname v→v,isae v→namof v,UU],
LOCOFVAR = [λv f s.isname v→v,isae v→arloc(namof v,VAL(subof v,f,s)),UU],
TYPOFVAR = [λv f s.isnamo v→TYPEOF(v,f,s),isae v→typolof(TYPEOF(namof v,f,s)),UU],
EXPRFORV = [λv f s.isname v→v,isae v→mkae(namof v,EXPRVAL(subof v)),UU],
VARBNDTO = [λv f s.ISBND(NAMOFVAR v,f,s)→
                  isname v → BVALOF(v,f,s),
                  isae v   → mkae(BVALOF(namof v,f,s),subof v).UU,v],

ISINTYPE  = [λv val f s.ISLOCAL(typeloc NAMOFVAR(v),s(f)) →
                        ISADMISVAL(TYPOFVAR(v,f,s),val(s)),FF],

ISADMISVAL = [λty v.(ty=INT)→isint v,issublim ty→ISINBOUND(v,ty),UU],

ISINBOUND = [∝F.[λx y.
              iseof x  → TT,
              ispair x → F(fstof x,fstof y)∧F(rmdof x,rmdof y),
              isint x  → issublim y→(x≥numof(lbof y))∧(x≤numof(ubof y)),UU,UU]],

VAL       = [∝F.[λp f s.
              iseof p  → EOF,
              ispair p → mkpair(MEXPR(fstof p,f,s),F(rmdof p,f,s)),UU]],

EXPRVAL   = [∝F.[λp f s.
              iseof p  → EOF,
              ispair p → mkpair(mknumconst(MEXPR(frstof p,f,s)),F(rmdof p,f,s)),UU]];
```

3.8 The Lookup of the Store

AXIOM LOOKUP:

```
IBUFFER ≡ [λs.s(0,fileloc INP)],
OBUFFER ≡ [λs.s(0,fileloc OUT)],
TEXT    ≡ [λf s.s(f,textloc)],
PROCDEF ≡ [λn f s.FETCH(procloc n,f,s)],
FUNCDEF ≡ [λn f s.FETCH(funcloc n,f,s)],
TYPEDEF ≡ [λloc f s.FETCH(loc,f,s)],
PROCTXT ≡ [λn f s.textof(PROCDEF(n,f,s))],
FUNCTXT ≡ [λn f s.textof(FUNCDEF(n,f,s))],
PROCFAL ≡ [λn f s.fargof(PROCDEF(n,f,s))],
FUNCFAL ≡ [λn f s.fargof(FUNCDEF(n,f,s))],
PROCBODY ≡ [λn f s.statmof(PROCTXT(n,f,s))],
FUNCBODY ≡ [λn f s.statmof(FUNCTXT(n,f,s))],
PROCDECL ≡ [λn f s.declof(PROCTXT(n,f,s))],
FUNCDECL ≡ [λn f s.declof(FUNCTXT(n,f,s))],

PFLNK   ≡ [λn f s. FETCH(acclnk n,f,s)],
NEWFP   ≡ [λn f s. ISBND(NAMOFVAR v,f,s)→ pred f,CRNTF(f,s)],
CRNTF   ≡ [λf s. s(f,alnk)],
FETCH   ≡ [∝F.[λl f s.ISLOCAL(l,s(f))→s(f,l),istopf(f)→UU,F(l,CRNTF(f,s),s)]],
FETCHV  ≡ [∝F.[λn f s.ISLOCAL(typeloc NAMOFVAR(n),s(f))→
                    ISLOCAL(NAMOFVAR(n),s(f))→s(f,LOCOFVAR(n,f,s)),UU,
                    istopf(f)→UU,F(VARBNDTO(n,f,s),NEWFP(n,f,s),s)]],

TYPEOF  ≡ [λn f s.s(f,typeloc n)],
BVALOF  ≡ [λn f s.s(f,bindloc n)];
```

3.9 Updating and Miscellaneous Axioms

AXIOM UPDATE:

```
STORE    ≡ [λf s loc val.[λf1.f1=f→MODFRAME(s(f),loc,val),s(f1)]],
MODFRAME ≡ [λf loc val.[λloc1.loc1=loc→val,f(loc1)]],
MAKFRAME ≡ [λtxt ln f s.[λf1.f1=f→[λloc1.loc1=textloc→txt,loc1=alnk → ln,UNDEF],s(f1)]];
```

AXIOM FRAME:

```
frame  ≡ [λs f.s(f)],
istopf ≡ [λf.(f=0)];
```

AXIOM AUXSEM:

```
!⊗     ≡ [λf g r.g(f(r))],
ID     ≡ [λx.x],
ISFUNFR ≡ [∝F.[λf s nf. ISLOCAL(FUNV,s(f))→ FF,pred f=nf → TT,F(pred f,s,nf)]],
ISLOCAL ≡ [λloc fr.fr(loc)=UNDEF→FF,TT],
ISPRESENT ≡ [λn fr.isname n→ISLOCAL(typidloc n,fr)∨ISLOCAL(typeloc n,fr)∨
                              ISLOCAL(acclnk n,fr)∨ISLOCAL(bindloc n,fr),UU],
RETURN ≡ [λf s.ISLOCAL(FUNV,s(f))→s(f,FUNV),UU],
ISBND  ≡ [λn s f.ISLOCAL(bindloc n,s(f))];
```

APPENDIX 4

Proof of the equivalence involving WHILE for goto-free programs

4.1 List of LCF commands

```
TRY 1 INDUCT -;
TRY 1 SPREF;

LABEL INDUCT;
TRY 2 SPREF;
USE GOTOF;
APPL - ,S,mkcmpnd(mkwhile(t,S),ES),f;
LABEL  HELP;
SIMPL - BY GOTOF1;
APPL .INDUCT,S,t,f;
SIMPL -;
USE CONO1 -;
APPL -,MBEXPR(t,f),S;
SIMPL -;
SS+.HELP;

   TRY SSUBST .MSFP OCC 3;
   TRY ;OED;
```

4.2 Printout of the proof

```
--------------------
|TRY #1   ∀S t f . isgotofree(S) :: WHILE(MBEXPR(t,f),MS(S,f)) c MS(mkcmpnd(mkwhile(t,S),ES),f)      INDUCT 331 ~
| --------------------
| |TRY #1#1   ∀S t t . isgotofree(S) :: UU(MBEXPR(t,f),MS(S,t)) c MS(mkcmpnd(mkwhile(t,S),ES),t)      SPREF.
| |332   ∀S t t . isgotofree(S) :: UU(MBEXPR(t,t),MS(S,t)) c MS(mkcmpnd(mkwhile(t,S),ES),f)    --- SPREF BY .
| --------------------
| --------------------
| |TRY #1#2   ∀S t t . isgotofree(S) :: [λt b .CONO(t,b↦F(t,b),ID)](MBEXPR(t,t),MS(S,f)) c MS(mkcmpnd(mkwhile(t~
,S),ES),f) : ASSUME   ∀S t t . isgotofree(S) :: F(MBEXPR(t,f),MS(S,t)) c MS(mkcmpnd(mkwhile(t,S),ES),t)      SPRE~
F.
| |333   ∀S t f . isgotofree(S) :: F(MBEXPR(t,f),MS(S,f)) c MS(mkcmpnd(mkwhile(t,S),ES),f)  (333) --- ASSUME.
| |334   isgotofree(S) ≡ TT  (334) --- SASSUME.
| |335   ∀S P f . isgotofree(S) :: isgotofree(P) :: MS(append(S,P),t) ≡ MS(S,f)⊗MS(P,t)    --- USE GOTOF.
| |336   [λS P f .isgotofree(S)→(isgotofree(P)→MS(append(S,P),t),UU),UU](S,mkcmpnd(mkwhile(t,S),ES),t) ≡ [λS P ~
t .isgotofree(S)→(isgotofree(P)→(MS(S,f)⊗MS(P,f)),UU),UU](S,mkcmpnd(mkwhile(t,S),ES),t)    --- APPL 335 S mkcmpnd~
(mkwhile(t,S),ES) t.
| |337   MS(append(S,mkcmpnd(mkwhile(t,S),ES)),f) ≡ MS(S,f)⊗MS(mkcmpnd(mkwhile(t,S),ES),f)  (334) --- SIMPL 336~
BY 33  GOTOF1 .
| |338   [λS t f .isgotofree(S)→F(MBEXPR(t,f),MS(S,t)),UU](S,t,f) c [λS t f .isgotofree(S)→MS(mkcmpnd(mkwhile(t~
,S),ES),f),UU](S,t,f)  (333) --- APPL 333 S t f.
| |339   F(MBEXPR(t,t),MS(S,t)) c MS(mkcmpnd(mkwhile(t,S),ES),f)  (333 334) --- SIMPL 338 BY 334 .
| |340   ∀T S1 . CONO(T,MS(S1,t)⊗F(MBEXPR(t,f),MS(S,f)),ID) c CONO(T,MS(S1,f)⊗MS(mkcmpnd(mkwhile(t,S),ES),f),ID~
)  (333 334) --- USE CONO1 339.
| |341   [λT S1 .CONO(T,MS(S1,t)⊗F(MBEXPR(t,t),MS(S,f)),ID)](MBEXPR(t,f),S) c [λT S1 .CONO(T,MS(S1,t)⊗MS(mkcmpn~
d(mkwhile(t,S),ES),f),ID)](MBEXPR(t,f),S)  (333 334) --- APPL 340 MBEXPR(t,t) S.
| |342   CONO(MBEXPR(t,t),MS(S,t)⊗F(MBEXPR(t,t) MS(S,f)),ID) c CONO(MBEXPR(t,t),MS(S,f)⊗MS(mkcmpnd(mkwhile(t,S)~
,ES),f),ID)  (333 334) --- SIMPL 341.
| | --------------------
| | |TRY #1#2#1   CONO(MBEXPR(t,f),MS(S,f)⊗F(MBEXPR(t,f),MS(S,t)),ID) c MS(mkcmpnd(mkwhile(t,S),ES),f)      SSUB~
ST 320 OCC 3.
| | | --------------------
| | | |TRY #1#2#1#1   CONO(MBEXPR(t,t),MS(S,t)⊗F(MBEXPR(t,t),MS(S,f)),ID) c CONO(MBEXPR(t,t),MS(S,t)⊗MS(mkcmpnd~
(mkwhile(t,S),ES),f),ID)    .
| | | --------------------
| | |343   CONO(MBEXPR(t,f),MS(S,t)⊗F(MBEXPR(t,f),MS(S,f)),ID) c MS(mkcmpnd(mkwhile(t,S),ES),f)  (333 334) --- ~
SSUBST 342 USING 320 OCC 3.
| | --------------------
| |344   ∀S t t . isgotofree(S) :: [λt b .CONO(t,b↦F(t,b),ID)](MBEXPR(t,t),MS(S,f)) c MS(mkcmpnd(mkwhile(t,S),E~
S),t)  (333) --- SPREF 343.
| --------------------
|345   ∀S t f . isgotofree(S) :: WHILE(MBEXPR(t,t),MS(S,f)) c MS(mkcmpnd(mkwhile(t,S),ES),f)   --- INDUCT 332 3~
44.
--------------------
```

APPENDIX 5

Proof of the equivalence involving REPEAT for goto-free programs

5.1 List of LCF commands

```
TRY 1 INDUCT -;
TRY 1 SPREF;

LABEL INDUCT;
TRY 2 SPREF;
USE GOTOF;
APPL - ,S,mkcmpnd(mkcond(mkbexpr1(not,t),mkrepeat(S,t),ES),ES),f ;
LABEL  HELP;
SIMPL - BY GOTOF1;
APPL .INDUCT,S,t,f;
SIMPL -;
USE COND1 -;
APPL -,MBEXPR(mkbexpr1(not,t),f),S;
SIMPL -;
SS+.HELP;

TRY SSUBST .MSFP OCC 3;
TRY SSUBST .MSFP OCC 4;
TRY ;QED;
```

5.2 Printout of the proof

```
 --------------------
 |TRY #1   ∀S t f . isgotofree(S) :: REPEAT(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)~
    INDUCT 331 .
 | --------------------
 | |TRY #1#1   ∀S t f . isgotofree(S) :: UU(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)~
    SPREF.
 | |332   ∀S t f . isgotofree(S) :: UU(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)   ---~
- SPREF BY .
 | --------------------
 | --------------------
 | |TRY #1#2   ∀S t f . isgotofree(S) :: ]λb t .beCOND(t,F(b,t),]D)][MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkc~
mpnd(mkrepeat(S,t),ES),f) : ASSUME   ∀S t f . isgotofree(S) :: F(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd~
(mkrepeat(S,t),ES),f)     SPREF.
 | |333   ∀S t f . isgotofree(S) :: F(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)   (333~
) --- ASSUME.
 | |334   isgotofree(S) ≡ TT   (334) --- SASSUME.
 | |335   ∀S P f . isgotofree(S) :: isgotofree(P) :: MS(append(S,P),f) ≡ MS(S,f)&MS(P,f)   --- USE GOTOF.
 | |336   ]λS P f .isgotofree(S)→(isgotofree(P)→MS(append(S,P),f),UU),UU](S,mkcmpnd(mkcond(mkbexprl(not,t),mkrep~
eat(S,t),ES),ES),f) ≡ ]λS P f .isgotofree(S)→(isgotofree(P)→(MS(S,f)&MS(P,f)),UU),UU](S,mkcmpnd(mkcond(mkbexprl(~
not,t),mkrepeat(S,t),ES),ES),f)   --- APPL 335 S mkcmpnd(mkcond(mkbexprl(not,t),mkrepeat(S,t),ES),ES) f.
 | |337   MS(append(S,mkcmpnd(mkcond(mkbexprl(not,t),mkrepeat(S,t),ES),ES)),f) ≡ MS(S,f)&MS(mkcmpnd(mkcond(mkbex~
prl(not,t),mkrepeat(S,t),ES),ES),f)   (334) --- SIMPL 336 BY 334 GDTDF1 .
 | |338   ]λS t f .isgotofree(S)→F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),UU](S,t,f) c [λS t f .isgotofree(S)→MS(mkc~
mpnd(mkrepeat(S,t),ES),f),UU](S,t,f)   (333) --- APPL 333 S t f.
 | |339   F(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)   (333 334) --- SIMPL 338 BY 334~
 .
 | |340   ∀T S1 . MS(S1,f)&COND(T,F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),ID) c MS(S1,f)&COND(T,MS(mkcmpnd(mkrepeat~
(S,t),ES),f),IO)   (333 334) --- USE COND1 339.
 | |341   ]λT S1 .MS(S1,f)&COND(T,F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),ID)](MBEXPR(mkbexprl(not,t),f),S) c ]λT S~
1 .MS(S1,f)&COND(T,MS(mkcmpnd(mkrepeat(S,t),ES),f),ID)](MBEXPR(mkbexprl(not,t),f),S)   (333 334) --- APPL 340 MBE~
XPR(mkbexprl(not,t),f) S.
 | |342   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),]D) c MS(S,f)&COND(MBEXPR(~
mkbexprl(not,t),f),MS(mkcmpnd(mkrepeat(S,t),ES),f),]D)   (333 334) --- SIMPL 341.
 | | --------------------
 | | |TRY #1#2#1   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),IO) c MS(mkcmpnd(~
mkrepeat(S,t),ES),f)     SSUBST 320 OCC 3.
 | | | --------------------
 | | | |TRY #1#2#1#1   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),]O) c MS(S,f)~
&MS(mkcmpnd(mkcond(mkbexprl(not,t),mkrepeat(S,t),ES),ES),f)     SSUBST 320 OCC 4.
 | | | | --------------------
 | | | | |TRY #1#2#1#1#1   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),]O) c MS(~
S,f)&COND(MBEXPR(mkbexprl(not,t),f),MS(mkcmpnd(mkrepeat(S,t),ES),f),]D)     .
 | | | | --------------------
 | | | |343   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),ID) c MS(S,f)&MS(mkcmp~
nd(mkcond(mkbexprl(not,t),mkrepeat(S,t),ES),ES),f)   (333 334) --- SSUBST 342 USING 320 OCC 4.
 | | | --------------------
 | | |344   MS(S,f)&COND(MBEXPR(mkbexprl(not,t),f),F(MS(S,f),MBEXPR(mkbexprl(not,t),f)),]D) c MS(mkcmpnd(mkrepea~
t(S,t),ES),f)   (333 334) --- SSUBST 343 USING 320 OCC 3.
 | | --------------------
 | |345   ∀S t f . isgotofree(S) :: ]λb t .b&COND(t,F(b,t),IO)](MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(~
mkrepeat(S,t),ES),f)   (333) --- SPREF 344.
 | --------------------
 |346   ∀S t f . isgotofree(S) :: REPEAT(MS(S,f),MBEXPR(mkbexprl(not,t),f)) c MS(mkcmpnd(mkrepeat(S,t),ES),f)   ~
--- ]NDUCT 332 345.
 --------------------
```

APPENDIX 6

Proof of the equivalence involving FORTO for goto-free programs

6.1 List of LCF commands

```
TRY 1 INDUCT -;
TRY 1 SPREF;

LABEL INDUCT;
TRY 2 SPREF;
USE GOTOF;
APPL - ,S,mkcmpnd(mkforto(i,mkexpr1(plus1,i),e2,S),ES) ,f;
LABEL  HELP;
SIMPL - ;
APPL .INDUCT,S,i,mkexpr1(plus1,i),e2,f;
SIMPL -;
USE COND1 -;
APPL -,MBEXPR(mkrel(leq,e,e2),f),S,ASSIGN(i,MEXPR(e,f),f);
SIMPL -;
SS+.HELP;

TRY SSUBST .MSFP OCC 3;
TRY ;QED;
```

6.2 Printout of the proof

```
 --------------------
 |TRY #1   ∀S i e1 e2 f . isgotofree(S) :: FDRTDf(i,e1,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e1,e2,S),ES),f)      I~
NDUCT 304 .
 | --------------------
 | |TRY #1#1   ∀S i e1 e2 f . isgotofree(S) :: UU(i,e1,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e1,e2,S),ES),f)      ~
SPREF.
 | |305   ∀S i e1 e2 f . isgotofree(S) :: UU(i,e1,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e1,e2,S),ES).f)   --- SPR~
EF BY .
 | --------------------
 | --------------------
 | |TRY #1#2   ∀S i e e2 f . isgotofree(S) :: [λi e e2 b f .COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPR(e,f)~
,f)∗b)∗F(i,mkexpr1(plus1,i),e2,b,f),ID)](i,e,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e,e2,S),ES),f) : ASSUME   ∀S i~
e e2 f . isgotofree(S) :: F(i,e,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e,e2,S),ES),f)      SPREF.
 | |306   ∀S i e e2 f . isgotofree(S) :: F(i,e,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e,e2,S),ES),f)   (306) --- AS~
SUME.
 | |307   isgotofree(S) ≡ TT   (307) --- SASSUME.
 | |308   ∀S P f . isgotofree(S) :: isgotofree(P) :: MS(append(S,P),f) ≡ MS(S,f)∗MS(P,f)   --- USE GDTDF.
 | |309   [λS P f .isgotofree(S)→(isgotofree(P)→MS(append(S,P),f),UU),UU](S,mkcmpnd(mkforto(i,mkexpr1(plus1,i),e~
2,S),ES),f) ≡ [λS P f .isgotofree(S)→(isgotofree(P)→(MS(S,f)∗MS(P,f)),UU),UU](S,mkcmpnd(mkforto(i,mkexpr1(plus1,~
i),e2,S),ES),f)   --- APPL 308 S mkcmpnd(mkforto(i,mkexpr1fplus1,i),e2,S),ES) f.
 | |310   MS(append(S,mkcmpnd(mkforto(i,mkexpr1(plus1,i),e2,S),ES)),f) ≡ MS(S,f)∗MS(mkcmpnd(mkforto(i,mkexpr1(pl~
us1,i),e2,S),ES),f)   (307) --- SIMPL 309 BY 307 GDTDF1 .
 | |311   [λS i e e2 f .isgotofree(S)→Ff(i,e,e2,MS(S,f),f),UU](S,i,mkexpr1(plus1,i),e2,f) ⊂ [λS i e e2 f .isgotof~
ree(S)→MS(mkcmpnd(mkforto(i,e,e2,S),ES),f),UU](S,i,mkexpr1(plus1,i),e2,f)   (306) --- APPL 306 S i mkexpr1(plus1,~
i) e2 f.
 | |312   Ff(i,mkexpr1(plus1,i),e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,mkexpr1fplus1,i),e2,S),ES),f)   (306 307) ---~
 SIMPL 311 BY 307 .
 | |313   ∀T S1 H . COND(T,(H∗MS(S1,f))∗F(i,mkexpr1(plus1,i),e2,MS(S,f),f),ID) ⊂ COND(T,H∗(MS(S1,f)∗MS(mkcmpnd(m~
kforto(i,mkexpr1(plus1,i),e2,S),ES),f)),ID)   (306 307) --- USE COND1 312.
 | |314   [λT S1 H .COND(T,(H∗MS(S1,f))∗F(i,mkexpr1(plus1,i),e2,MS(S,f),f),ID)](MBEXPR(mkrel(lseq,e,e2),f),S,ASS~
IGN(i,MEXPR(e,f),f)) ⊂ [λT S1 H .COND(T,H∗(MS(S1,f)∗MS(mkcmpnd(mkforto(i,mkexpr1(plus1,i),e2,S),ES),f)),ID)](MBE~
XPR(mkrel(lseq,e,e2),f),S,ASSIGN(i,MEXPR(e,f),f))   (306 307) --- APPL 313 MBEXPR(mkrel(lseq,e,e2),f) S ASSIGN(i,~
MEXPR(e,f),f).
 | |315   COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPR(e,f),f)∗MS(S,f))∗F(i,mkexpr1(plus1,i),e2,MS(S,f),f),ID~
) ⊂ COND(MBEXPR(mkrel(lseq,e,e2),f),ASSIGN(i,MEXPR(e,f),f)∗(MS(S,f)∗MS(mkcmpnd(mkforto(i,mkexpr1(plus1,i),e2,S),~
ES),f)),ID)   (306 307) --- SIMPL 314.
 | | --------------------
 | | |TRY #1#2#1   COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPR(e,f),f)∗MS(S,f))∗F(i,mkexpr1(plus1,i),e2,MS(S~
,f),f),ID) ⊂ MS(mkcmpnd(mkforto(i,e,e2,S),ES),f)      SSUBST 293 OCC 3.
 | | | --------------------
 | | | |TRY #1#2#1#1   COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPRfe,f),f)∗MS(S,f))∗F(i,mkexpr1(plus1,i),e2,~
MS(S,f),f),ID) ⊂ COND(MBEXPR(mkrel(lseq,e,e2),f),ASSIGN(i,MEXPR(e,f),f)∗(MSfS,f)∗MSfmkcmpnd(mkforto(i,mkexpr1(pl~
us1,i),e2,S),ES),f)),ID)     .
 | | | --------------------
 | | |316   COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPR(e,f),f)∗MS(S,f))∗F(i,mkexpr1(plus1,i),e2,MSfS,f),f),~
ID) ⊂ MSfmkcmpnd(mkforto(i,e,e2,S),ES),f)   (306 307) --- SSUBST 315 USING 293 OCC 3.
 | | --------------------
 | |317   ∀S i e e2 f . isgotofree(S) :: [λi e e2 b f .COND(MBEXPR(mkrel(lseq,e,e2),f),(ASSIGN(i,MEXPR(e,f),f)∗b~
)∗F(i,mkexpr1(plus1,i),e2,b,f),ID)](i,e,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e,e2,S),ES),f)   (306) --- SPREF 316~
 | --------------------
 |318   ∀S i e1 e2 f . isgotofree(S) :: FDRTDf(i,e1,e2,MS(S,f),f) ⊂ MS(mkcmpnd(mkforto(i,e1,e2,S),ES),f)   --- IN~
DUCT 305 317.
 --------------------
```

APPENDIX 7

Proof of the goto-free factorial program

7.1 List of LCF commands

```
SS+ .APPLY,.FUNCT,.PASCAL,.MP,.FUNCCOMP,.ID,.DP,.SP,.MD;
TRY SIMPL;
TRY INDUCT .WHILE;

   TRY 1 SPREF;
   SS + .COND; SS - .SP;
   LABEL INDUCT;
   TRY 2 SPREF;
   LABEL L1 --;
   TRY CASES ¬(n=0);

      TRY 3 SIMPL;

      TRY 2;
      USE ARITH1 .L1 ,-;
      QED -;

      TRY 1 SIMPL;
      APPL .INDUCT,pred n,x*n;
      SIMPL -;
      TRY ; QED;
```

7.2 Printout of the proof

```
|TRY #1   ∀n x . isnat(n) :: isnat(x) :: APPLY(FACTORIAL,n,x) c FACT(n,x)     SIMPL.
| --------------------
| |TRY #1#1   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,WHILE(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(n~
2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x)      INDUCT 314 .
| | --------------------
| | |TRY #1#1#1   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,UU(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(~
n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x)      SPREF.
| | |318   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,UU(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(n2,0,CR~
EAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x) --- SPREF BY  TH8 TH6.
| | --------------------
| | --------------------
| | |TRY #1#1#2   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,[λt b .COND(t,b≠F(t,b),ID)](MBEXPR(test,0),M~
S(body,0),READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c ~
FACT(n,x) : ASSUME   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,F(MBEXPR(test,0),MS(body,0),READ(n1,0,READ~
(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x)      SPREF.
| | |319   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,F(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(n2,0,CRE~
AV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x)  (319) --- ASSUME.
| | |320   isnat(n) ≡ TT  (320) --- SASSUME.
| | |321   isnat(x) ≡ TT  (321) --- SASSUME.
| | | --------------------
| | | |TRY #1#1#2#1   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(bo[illegible]),MS(body,0,FRAME1(SP,n,x))),FRAME1(SP~
,n,x))) c FACT(n,x)     CASES ¬(n=0).
| | | | --------------------
| | | | |TRY #1#1#2#1#3   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME~
1(SP,n,x))) c FACT(n,x) : SASSUME   ¬(n=0) ≡ FF     SIMPL.
| | | | |322   ¬(n=0) ≡ FF  (322) --- SASSUME.
| | | | |323   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME1(SP,n,x)))~
c FACT(n,x)  (321 322) --- SIMPL BY 321 322  LM4.
| | | | --------------------
| | | | --------------------
| | | | |TRY #1#1#2#1#2   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME~
1(SP,n,x))) c FACT(n,x) : SASSUME   ¬(n=0) ≡ UU     .
| | | | |324   ¬(n=0) ≡ UU  (324) --- SASSUME.
| | | | |325   TT ≡ UU  (320 324) --- USE ARITH1 320 324.
| | | | |326   TT ≡ UU  (320 324) --- INCL 325.
| | | | --------------------
| | | | --------------------
| | | | |TRY #1#1#2#1#1   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME~
1(SP,n,x))) c FACT(n,x) : SASSUME   ¬(n=0) ≡ TT     SIMPL.
| | | | |327   ¬(n=0) ≡ TT  (327) --- SASSUME.
| | | | |328   [λn x .isnat(n)→(isnat(x)→RESULT(WRITE(n1,0,F(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(n2,0,CREA~
V(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))),UU),UU](pred(n),x≠n) c [λn x .isnat(~
n)→(isnat(x)→FACT(n,x),UU),UU](pred(n),x≠n)  (319) --- APPL 319 pred(n) x≠n.
| | | | |329   RESULT(WRITE(n1,0,F(MBEXPR(test,0),MS(body,0),FRAME1(SP,pred(n),x≠n)))) c FACT(n,x)  (319 320 32~
1 327) --- SIMPL 328 BY 320 321 327 LM1 ARITH2 ARITH3  ARITH4.
| | | | | --------------------
| | | | | |TRY #1#1#2#1#1#1 RESULT(WRITE(n1,0,F(MBEXPR(test,0),MS(body,0),FRAME1(SP,pred(n),x≠n)))) c FACT(n,x).
| | | | | --------------------
| | | | |330   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME1(SP,n,x)))~
c FACT(n,x)  (319 320 321 327) --- SIMPL 329 BY 321 327  LM2.
| | | | --------------------
| | | |331   RESULT(WRITE(n1,0,¬(n=0)→F(MBEXPR(test,0),MS(body,0),MS(body,0,FRAME1(SP,n,x))),FRAME1(SP,n,x))) c~
FACT(n,x)  (319 320 321) --- CASES ¬(n=0) 330 326 323.
| | | --------------------
| | |332   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,[λt b .COND(t,b≠F(t,b),ID)](MBEXPR(test,0),MS(body,~
0),READ(n1,0,READ(n2,0,CREAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,~
x)  (319) --- SPREF 331 BY 227 280 281 320 321 LM3 LM1.
| | --------------------
| |333   ∀n x . isnat(n) :: isnat(x) :: RESULT(WRITE(n1,0,WHILE(MBEXPR(test,0),MS(body,0),READ(n1,0,READ(n2,0,C~
REAV(0,n2,INT,0,CREAV(0,n1,INT,0,FRAME0(FACTORIAL,INPUT(LIST(n,x)),EOF)))))))) c FACT(n,x)     --- INDUCT 318 332.
| --------------------
|334   ∀n x . isnat(n) :: isnat(x) :: APPLY(FACTORIAL,n,x) c FACT(n,x)     --- SIMPL 333 BY 207 208 210 214 280 2~
81 304 306 307 310 311 316 TH13 TH15 TH10 TH12 TH11 TH5 TH14 TH2 TH7 TH3 TH1.
```

APPENDIX 8

Proof of the McCarthy Airline Reservation System

8.1 List of LCF commands

```
SS+ .APPLY,.FUNCT,.PASCAL,.FUNCCOMP,.MP,.SP;
TRY SIMPL;
TRY INDUCT .REPEAT;
TRY 1 SPREF;
TRY CASES ¬(ell(isq)=3);
   TRY 3 SIMPL;
   TRY 2; USE ARITH1 -,-------; QED;
   TRY 1 SIMPL;
LABEL INDUCT;
TRY 2 SPREF;
TRY CASES ¬(ell(isq)=3);
   TRY 3 SIMPL;
   TRY 2; USE ARITH1 -,-------; QED;
   SS+ .COND, .ID;
   TRY 1 SIMPL;
   APPL .INDUCT,tail isq,mkpair(stupdt(isq,p,q),osq),stupdt(isq,p,q),wiupdt(isq,p,q);
   SIMPL -;
   TRY; QED;
```

8.2 Printout of the proof

```
 --------------------
 |TRY #1   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: APPLY(McCARTHY,p,q,isq,osq) c BO~
OKING(p,q,isq,osq)      SIMPL.
 | ----------  --------
 | |TRY #1#1   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,~
0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→REPEAT(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3)~
)),0),0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))~
))) c BOOKING(p,q,isq,osq)      INDUCT 308 .
 | | --------------------
 | | |TRY #1#1#1   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(B~
ODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→UU(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3)~
)),0),0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))~
))) c BOOKING(p,q,isq,osq)      SPREF.
 | | |335   iswfsq(isq) ≡ TT  (335) --- SASSUME.
 | | |336   iswfos(osq) ≡ TT  (336) --- SASSUME.
 | | |337   isint(p) ≡ TT  (337) --- SASSUME.
 | | |338   isint(q) ≡ TT  (338) --- SASSUME.
 | | | --------------------
 | | | |TRY #1#1#1#1   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq)      CASES ~
¬(ell(isq)=3).
 | | | | --------------------
 | | | | |TRY #1#1#(#1#3   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASS~
UME   ¬(ell(isq)=3) ≡ FF      SIMPL.
 | | | | |339   ¬(ell(isq)=3) ≡ FF  (339) --- SASSUME.
 | | | | |340   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq)  (335 336 337 338~
 339) --- SIMPL BY 335 336 337 338 339  LM3.
 | | | | ----------  --------
 | | | | --------------------
 | | | | |TRY #1#1#1#1#2   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASS~
UME   ¬(ell(isq)=3) ≡ UU      .
 | | | | |341   ¬(ell(isq)=3) ≡ UU  (341) --- SASSUME.
 | | | | |342   TT ≡ UU  (335 341) --- USE ARITH1 341 335.
 | | | | --------------------
 | | | | --------------------
 | | | | |TRY #1#1#1#1#1   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASS~
UME   ¬(ell(isq)=3) ≡ TT      SIMPL.
 | | | | |343   ¬(ell(isq)=3) ≡ TT  (343) --- SASSUME.
 | | | | |344   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq)  (343) --- SIMPL ~
BY 343  THG.
 | | | | --------------------
 | | | |345   OUTPUT(¬(ell(isq)=3)→UU,MS(BODY,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq)  (335 336 337 338) ~
--- CASES ¬(ell(isq)=3) 344 342 340.
 | | | --------------------
 | | |346   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,R~
EAD(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→UU(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3))),0),0~
,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))) c B~
OOKING(p,q,isq,osq)    --- SPREF 345 BY 335 336 337 338 LM9  LM2.
 | | --------------------
 | | --------------------
 | | |TRY #1#1#2   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(B~
ODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→(λB T f .B&COND(T,F(B,T,f),ID))(MS(BODY,0),MBEXPR(mkbexpr1(~
not,mkrel(eq,rq,mknumconst(3))),0),0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,R~
EAD(wl,0,FRAME1(p,q,isq,osq))))) c BOOKING(p,q,isq,osq)      :      ASSUME      ∀ isq osq p q . iswfsq(i~
sq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p~
,q,isq,osq))))=3)→F(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumcons((3))),0),0,MS(BODY,0,READ(st,0,READ(wl~
,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))) c BOOKING(p,q,isq,osq)      SPRE~
F.
 | | |347   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,R~
EAD(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→F(MS(BODY,0),MBEXPR(mkbexpr1(not,mkrel(eq,rq,mknumconst(3))),0),0,~
MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq)))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))) c BO~
OKING(p,q,isq,osq)  (347) --- ASSUME.
 | | |348   iswfsq(isq) ≡ TT  (348) --- SASSUME.
 | | |349   iswfos(osq) ≡ TT  (349) --- SASSUME.
 | | |350   isint(p) ≡ TT  (350) --- SASSUME.
```

```
| | |351   isint(q) ≡ TT  (351) --- SASSUME.
| | | --------------------
| | | |TRY #1#1#2#1   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BODY,0)~
,MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY,0,F~
RAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq)     CASES ¬(ell(isq)=3).
| | | | --------------------
| | | | |TRY #1#1#2#1#3   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BOD~
Y,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY~
,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASSUME   ¬(ell(isq)=3) ≡ FF     SIMPL.
| | | | |352   ¬(ell(isq)=3) ≡ FF  (352) --- SASSUME.
| | | | |353   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BODY,0),MBEXPR~
(mkbexprl(not,mkrel(eq,rq,mknumconst(3))) 0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY,0,FRAME2(p~
,q,isq,osq))) c BOOKING(p,q,isq,osq)  (348 349 350 351 352) --- SIMPL BY 348 349 350 351 352  LM3.
| | | | --------------------
| | | | --------------------
| | | | |TRY #1#1#2#1#2   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BOD~
Y,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY~
,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASSUME   ¬(ell(isq)=3) ≡ UU     .
| | | | |354   ¬(ell(isq)=3) ≡ UU  (354) --- SASSUME.
| | | | |355   TT ≡ UU  (348 354) --- USE ARITH1 354 348.
| | | | --------------------
| | | | --------------------
| | | | |TRY #1#1#2#1#1   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BOD~
Y,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY~
,0,FRAME2(p,q,isq,osq))) c BOOKING(p,q,isq,osq) : SASSUME   ¬(ell(isq)=3) ≡ TT     SIMPL.
| | | | |356   ¬(ell(isq)=3) ≡ TT  (356) --- SASSUME.
| | | | |357   (λisq osq p q . iswfsq(isq)→(iswfos(osq)→(isint(p)→(isint(q)→OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,READ(s~
t,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→F(MS(BODY,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0,MS(BO~
DY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))),MS(BODY,0,REAO(st,0,REAO(wl,0,FRAME1(p,q,isq,osq))))),UU),UU),U~
U),UU)(tail(isq),mkpair(stupdt(isq,p,q),osq),stupdt(isq,p,q),wlupdt(isq,p,q)) c (λisq osq p q . iswfsq(isq)→(isw~
fos(osq)→(isint(p)→(isint(q)→BOOKING(p,q,isq,osq),UU),UU),UU),UU)(tail(isq),mkpair(stupdt(isq,p,q),osq),stupdt(~
isq,p,q),wlupdt(isq,p,q))  (347) --- APPL 347 tail(isq) mkpair(stupdt(isq,p,q),osq) stupdt(isq,p,q) wlupdt(isq,~
p,q).
| | | | |358   OUTPUT(¬(el3(isq)=3)→F(MS(BODY,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0,MS(BODY,0~
,FRAME3(p,q,isq,osq))),MS(BODY,0,FRAME3(p,q,isq,osq))) c BOOKING(p,q,isq,osq)  (347 348 349 350 351 356) --- SIM~
PL 357 BY 348 349 350 351 356 LM7 LM2 LM5 ARITH2 ARITH3 ARITH4 ARITH5  LM4.
| | | | | --------------------
| | | | | |TRY #1#1#2#1#1#1   OUTPUT(¬(el3(isq)=3)→F(MS(BODY,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3)))~
,0),0,MS(BODY,0,FRAME3(p,q,isq,osq))),MS(BODY,0,FRAME3(p,q,isq,osq))) c BOOKING(p,q,isq,osq)     .
| | | | | --------------------
| | | | |359   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BODY,0),MBEXPR~
(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY,0,FRAME2(p~
,q,isq,osq))) c BOOKING(p,q,isq,osq)  (347 348 349 350 351 356) --- SIMPL 358 BY 227 281 348 349 350 351 356  LM~
8 LM6.
| | | | --------------------
| | | |360   OUTPUT(¬(ell(isq)=3)→CONO(MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0),F(MS(BODY,0),MBEXPR(m~
kbexprl(not,mkrel(eq,rq,mknumconst(3))),0),0),IO,MS(BODY,0,MS(BODY,0,FRAME2(p,q,isq,osq)))),MS(BODY,0,FRAME2(p,q~
,isq,osq))) c BOOKING(p,q,isq,osq)  (347 348 349 350 351) --- CASES ¬(ell(isq)=3) 359 355 356.
| | | --------------------
| | |361   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,R~
EAO(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→(λB T f .B→CONO(T,F(B,T,f),IO)|(MS(BODY,0),MBEXPR(mkbexprl(not,mkre~
l(eq,rq,mknumconst(3))),0),0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))),MS(BODY,0,REAO(st,0,READ(wl,~
0,FRAME1(p,q,isq,osq))))) c BOOKING(p,q,isq,osq)  (347) --- SPREF 360 BY 280 348 349 350 351 LM9 LM2.
| | --------------------
| |362   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: OUTPUT(¬(MEXPR(rq,0,MS(BODY,0,REA~
D(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))=3)→REPEAT(MS(BODY,0),MBEXPR(mkbexprl(not,mkrel(eq,rq,mknumconst(3))),0)~
,0,MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))),MS(BODY,0,READ(st,0,READ(wl,0,FRAME1(p,q,isq,osq))))) c~
BOOKING(p,q,isq,osq)   --- INDUCT 346 361.
| --------------------
|363   ∀isq osq p q . iswfsq(isq) :: iswfos(osq) :: isint(p) :: isint(q) :: APPLY(McCARTHY,p,q,isq,osq) c BOOKI~
NG(p,q,isq,osq)   --- SIMPL 362 BY 207 208 210 280 303 326 333 334 LM1 TH2 TH5.
--------------------
```

REFERENCES

Aiello, L. and Aiello, M.,
1974 *Proving Program Correctness in LCF*,
Presented at the Colloquium on Programming, Paris, 9-11 April 1974.

Dijkstra, E.W.,
1972 *Notes on Structured Programming*,
Structured Programming, by Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R.,
Academic Press (1972) 1-82.

Floyd, R.W.,
1967 *Assigning meanings to programs*,
Proc. of a Symposium on Applied Mathematics, Vol. 19 - Mathematical Aspects of Science, (Schwartz, J.T. ed.), American Math. Society (1967) 19-32.

Haberman, A. M.,
1973 *Critical Comments on the Programming Language PASCAL*,
Department of Computer Science, Carnegie-Mellon University, October 1973.

Hoare, C.A.R.,
1969 *An Axiomatic Basis for Computer Programming*,
Comm. ACM, Vol.12, No. 10 (1969) 576-580, 583.

Hoare, C.A.R.,
1972 *A note on the for statement*,
BIT 12 (1972) 334-341

Hoare, C.A.R. and Wirth, N.,
1973 *An Axiomatic Definition of the Programming Language PASCAL*,
Acta Informatica, Vol.2 (1973) 335-355.

Igarashi, S., London, R.L . and Luckham D.C.,
1973 *Automatic Program Verification I: A Logical Basis and its Implementation*,
Artificial Intelligence Memo No. 200, Stanford University (1973).

Kleene, S.C.,
1952 *Introduction to Metamathematics*
Van Nostrand Company Inc., New York 1952.

Kleene, S.C., and Vesley, R.E.
1965 *The Foundations of Intuitionistic Mathematics*
North-Holland Publishing Company, Amsterdam 1965.

Lucas, P. and Walk, K.,
1969 *On the Formal Description of PL/I,*
Ann. Rev. in Automatic Programming, Vol. 6, Part 3 (1969).

McCarthy, J.,
1961 *A Basis for a Mathematical Theory of Computation,*
Proc. of the Western Joint Comp. Conf., Spartan Books, New York (1961) 225-138.

McCarthy, J., and Painter, J.
1966 *Correctness of a compiler for arithmetic expressions*
Stanford Artificial Intelligence Memo No. 40 (1966).
Also in Math. Aspects of Computer Science, Am. Math. Soc. (1967).

Manna, Z.,
1969 *The correctness of programs,*
J.of Comp. and Sys. Science.,Vol.3 (1969) 119-127.

Milner, R.,
1972a *Logic for computable functions, description of a machine implementation*
Artificial Intelligence Memo No. 169,Stanford University (1972).

Milner, R.,
1972b *Implementation and Applications of Scott's Logic for Computable Functions,*
Proc. ACM Conf. on Proving Assertions about Programs.
New Mexico State University, Las Cruces, New Mexico (1972) 1-5.

Milner, R. and Weyhrauch, R.W.,
1972 *Proving Compiler Correctness in a Mechanized Logic,*
Machine Intelligence 7 (Meltzer, B. and Michie, D. Eds.),
Edinbourgh University Press (1972) 51-70.

Newey, M.,
1973 *Axioms and Theorems for Integers, Lists and Finite Sets in LCF,*
Artificial Intelligence Memo No.184, Stanford University (1973).

Newey, M.,
1974 *Formal Semantics of LISP with Applications to Program Correctness*
Forthcoming Ph. D. Dissertation, Stanford University, 1974.

Scott, D.S. and Strachey, C.,
1971 *Towards a Mathematical Semantics for Computer Languages,*
Proc. of the Symposium on Computers and Automata,
Microwave Research Institute Symposia Series, Vol.21,
Polytechnic Institute of Brooklyn (1971).

Scott, D.S.,
1971 *Continous Lattices, Proc. of 1971 Dalhousie Conf.,*
Springer Lecture Notes Series, Springer-Verlag, Heidelberg (1971).

Waldinger R. J. and Levitt K. N.,
1973 *Reasoning about programs*
TN 86: SRI-AI (1973)

Wirth, N.,
1971 *The Programming Language PASCAL,*
Acta Informatica, Vol.1 (1971) 35-63.

Wirth, N.,
1972 *The Programming Language PASCAL (Revised Report),*
Berichte der Fachgr., Computer-Wissenschaften, Nr. 5, E.T.H., Zurich (1972).

Weyhrauch, R.W. and Milner, R.,
1972 *Program Semantics and Correctness in a Mechanized Logic,*
Proc. 1st USA-Japan Computer Conf., Tokyo (1972).